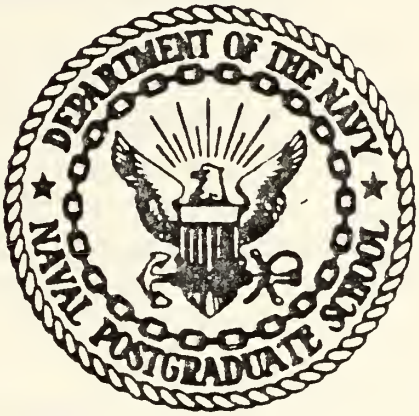
1977

A decision logic table preprocessor

Keller, Joseph Franklin

NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

A DECISION LOGIC TABLE

PREPROCESSOR

by

Joseph Franklin Keller
and
Robert William Roesch, Jr.

June 1977

Thesis Advisor: C. P. Gibfried

Approved for public release; distribution unlimited.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A DECISION LOGIC TABLE PREPROCESSOR	5. TYPE OF REPORT & PERIOD COVERED: Master's Thesis; June 1977
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Joseph Franklin Keller Robert William Roesch, Jr.	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Naval Postgraduate School Monterey, California 93940	12. REPORT DATE: June 1977 13. NUMBER OF PAGES: 141
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Naval Postgraduate School Monterey, California 93940	15. SECURITY CLASS. (of this report): Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Decision Logic Tables; Preprocessors; Computer Program Translators

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An introduction to decision logic table terminology, structure, theory, and preprocessor historical development. Advantages of decision tables and flowcharts have been surveyed and contrasted. Techniques of decision table preparation have been enumerated. DELTRANS, a rule mask logic table preprocessor for the C

language, has been proposed for use on the PDP-11/50 system at the U.S. Naval Postgraduate School.

A DECISION LOGIC TABLE PREPROCESSOR

by

Joseph Franklin Keller
Lieutenant, United States Navy
B.S., United States Naval Academy, 1969

and

Robert William Roesch, Jr.
Captain, United States Marine Corps
B.Ch.E., Georgia Institute of Technology, 1971

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the
NAVAL POSTGRADUATE SCHOOL
June 1977

ABSTRACT

An introduction to decision logic table terminology, structure, theory, and preprocessor historical development. Advantages of decision tables and flowcharts have been surveyed and contrasted. Techniques of decision table preparation have been enumerated. DELTRANS, a rule mask logic table preprocessor for the C language, has been proposed for use on the PDP-11/50 system at the U.S. Naval Postgraduate School.

CONTENTS

I. INTRODUCTION

A. BACKGROUND

A decision table preprocessor is a software program for translating decision logic tables into compilable source code. The preprocessor developed by the authors, aptly called DELTRANS, was designed to operate on the PDP-11/50 system at the Naval Postgraduate School and accept C language programs containing decision logic tables. The design of DELTRANS was based on the sequential testing rule mask technique perfected by Press [20].

The use of this decision table processor should reduce both programming effort and time. The additional computer time required for compilation is overshadowed by the reduction in manpower required for programming both during initial programming phases and maintenance phases. The concept and structure of decision logic tables causes the number of overlooked situations and program inconsistencies to be reduced.

Decision tables offer a stimulating alternative to traditional programming methods for those who are willing to educate themselves in their construction and use. With this knowledge, DELTRANS is but another tool for the C language programmer, possibly a very valuable one.

The major steps from program input to production of executable code are depicted in Figure 1. Initially, a file containing a C language program may be created at a terminal. The programmer codes those segments of the program not covered by decision logic tables. Special symbols indicate to the table preprocessor the beginning and ending of each table. Otherwise, the code is passed unchanged. At this point, a call to DELTRANS is initiated in order to produce compilable source code. As shown in Figure 1, a table listing may also be obtained. Subsequently, normal program compilation may be accomplished.

Appendix A, the DELTRANS User's Manual, was written as an independent document and as such, guides a user through the steps illustrated in Figure 1.

B. HISTORICAL DEVELOPMENT

1. Development of the First Processor

In the mid-1950s, General Electric's Manufacturing Services Department initiated a research effort to study the manufacturing processes that occur from the receipt of a customer order through the production of the finished product. Having recognized that computers might play a significant role, a search was begun to find a satisfactory method for expressing the complex logic encountered.

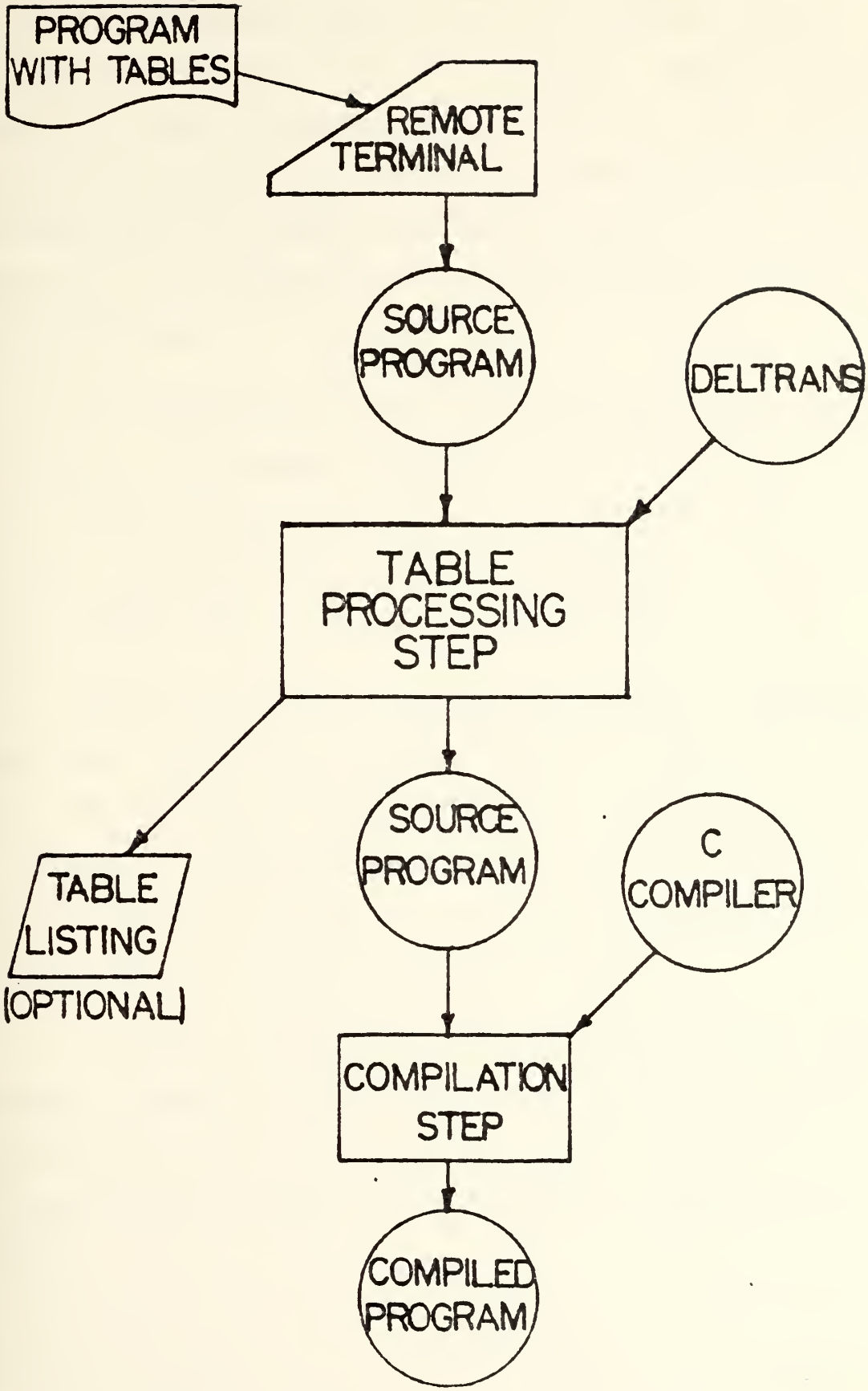


FIGURE 1. DELTRANS Functional Diagram

It soon became apparent that available methods of describing decisions, such as formulas, narratives, and flowcharts, were inadequate. The efforts of the project team to discover a new method of expression culminated in the development of "decision structure tables"[18]. These tables had a format similar to the truth tables from which they originated.

The processor for solving these tables, expressed in a language called TABSOL, operated initially on an IBM 702. Later it was successively implemented on an IBM 305, 650, and 704. In early 1961, an improved version of the processor and TABSOL were implemented on the GE 225.

During this same time period, Sutherland Management Consultants also began experimenting with decision tables [18]. They produced a table different in form but identical in concept. The emphasis was placed strictly on the use of decision tables as an aid to systems documentation, leaving the solution of the table to the programmer.

A number of companies, including Hunt Foods, North American Aviation, and the Insurance Company of North America, initiated research on decision tables [11], primarily to facilitate in-house file-maintenance and system documentation.

From 1960, the CODASYL systems study group has carried out work on the development of a high-level language, COBOL. Decision tables were selected as an addition to COBOL and in 1962, the specifications of DETAB-X were published. The manual described a decision table preprocessor that could accept the decision tables as input and produce a form of COBOL coding as output. This was the first table processor available to computer users.

2. Evolution and Refinement

In June 1965, the Special Interest Group for Programming Languages (SIGPLAN) of the Los Angeles Chapter of the Association of Computing Machinery appointed a working group to develop a decision table preprocessor. The result was the distribution of DETAB/65, written in a restricted subset of COBOL. Although implemented on the CDC 3600 and IBM 7094, among others, its inefficient conversion algorithm led to its demise.

It has become evident that DETAB/65 was, however, the ancestor of the current group of proprietary decision table preprocessors developed since 1966. Generally, the processors follow the DETAB/65 lead in that they are a preprocessor written in COBOL that converts decision tables containing COBOL components to a stream of COBOL source code suitable for compilation. There are several exceptions to this basic rule.

IBM's System/360 Decision Logic Translator processes decision tables coded in Fortran. There are also other notable examples using BASIC and ALGOL [15]. Some preprocessors offer the programmer the option of specifying the language to be processed.

3. Use Today

A review of decision logic table preprocessor history almost forces one to wonder why decision logic tables are not universally used for systems analysis, program development, and documentation. Many times, this technique has succeeded where the more widely used methods of narrative and flowcharting have failed. Why then has the use of decision logic tables not been more widespread?

Three possible causes have been identified. First, the amount of information available to systems analysts and programmers on a daily basis has been limited. Although many articles have been published over the years, they have generally appeared in highly technical form or have appeared in proceedings or journals not extensively circulated to the commercial practitioner. As a rule, decision tables have not been taught in their rightful place as an alternative to flowcharts in introductory programming courses.

Second, the use of decision tables requires a different approach to problem solving than does flowcharting.

Flowcharting leads one to adopt a sequential model of decision making. That is, a test followed by one or more actions. On the other hand, decision logic tables require an overall analysis of the conditions that comprise a given problem and the effect of their various combinations on the solution. Naturally, there is much resistance among those trained in sequential type analysis to accept a new technique.

Third, there has been a general lack of decision table processors available to the data processing community. As a result, tables had to be hand translated to sequential code for input to the computer. Absence of a mechanized means of translation has resulted in a rapid decrease of interest in decision tables by programmers.

These three conditions are slowly being eased with the increase of books and articles published on the subject. Seminars are available from several sources and presumably, the historical resistance to decision tables will be overcome.

C. FLOWCHARTING VERSUS DECISION TABLES

As has been pointed out, narratives and flowcharting have historically been used to document computer programs and structure their logic. These techniques have been demonstrated to be very effective time and time again, as

long as the problem remained relatively simple and straightforward. However, this effectiveness has severely deteriorated when the problem became more complex.

Decision logic tables have been shown to provide a format for organizing and displaying program logic, and have been compared with narratives and flowcharts in program documentation and logic structuring. This comparison has shown a number of important advantages and disadvantages of the use of decision logic tables.

One advantage of decision logic tables over other forms is the conciseness normally associated with a decision table. A much larger amount of information may be placed in a given space using a table as opposed to narratives or flowcharts. This dense display of information provides a much clearer representation of the program logic than a cloudy narrative or a branching, meandering flowchart.

The conciseness of decision logic tables leads directly to their second advantage. Namely, the advantage of thoroughness or completeness. The person preparing decision logic tables is forced by their format to consider all possible combinations of events. This is quite different from the flowcharting approach which tends to emphasize sequential logic flow. This emphasis on sequential logic flow often tends to obscure alternative logic, and may thoroughly avoid the issue of logic completeness. Failing to be

prepared for all possible combinations of events is of course a major source of subtle program "bugs".

Non-sequential logic flow also tends to assist the programmer in eliminating other logic errors. The elimination is due to the manner the entire flow of logic is displayed at a glance in the case of decision logic tables. The programmer is thus permitted to visualize better the interrelationships and alternatives within the problem at hand. Not only is completeness displayed but also redundant tests are pinpointed thus permitting the production of more efficient code.

Decision logic table construction and modification is easy to learn. Thus, a non-programmer can normally read the logic of a well written program. Further, dependency upon the original programmer is greatly reduced since modifications are easy to perform.

A final important advantage of decision logic tables over flowcharting is their ability to serve as computer input. This permits machine checking for certain types of logic errors and mechanized conversion into a program segment.

Several disadvantages to the user of decision logic tables do exist. Perhaps the most insidious is that the sequential flow of flowcharting has been the only technique

taught programmers. The effort to learn something "new" has only been reluctantly taken in may cases.

Another drawback is that although it is easy to learn to use decision logic tables, extensive work is required to become truly efficient in programming with them, as opposed to programming with flowcharts. When using a computer to convert the logic, further work is required to become familiar with the translator or preprocessor.

Even though the flow of logic is improved, the actions specified cannot be machine checked to ensure they are correct, or even feasible for that matter. Also, the machine is incapable of recognizing impossible combinations of events. This forces the programmer to perform these checks or supply some escape set of actions.

The advantages and disadvantages of decision logic tables discussed here have been summarized in Table 1. They may be compared with those of flowcharts that have been listed in Table 2.

Clearly, decision logic tables are not a panacea for all the ills of data processing today. However, they can be used as a very effective tool to ensure proper program logic flow and should be used in conjunction with narratives and flowcharts, as appropriate.

ADVANTAGES:

1. Clear enumeration of all operations performed.
2. Clear identification of the sequence of operations.
3. Easily learned.
4. Effective means of communication between people in and out of data processing.
5. Concise and compact form of documentation.
6. Easy to construct, modify and read.
7. Easy visualization of relationships and alternatives.
8. Directly adaptable to computer operations.

DISADVANTAGES:

1. May be large for complex situations.
2. Multiple tables may be needed.
3. Graphic display of flowcharts may be more meaningful.
4. Requirements too detailed for man-to-man communication.

TABLE 1. Advantages and Disadvantages of Decision Tables

ADVANTAGES:

1. Easily produced.
2. Easily learned.
3. Can describe data handling and computer operations.
4. Can be produced by computer algorithms from source programs.

DISADVANTAGES:

1. Heavily influenced by personal preference.
2. May be difficult to follow in complex problem.
3. Revision is difficult.
4. Limited in displaying all logical elements.
5. Detailed logic flowcharts are unwieldy.

TABLE 2. Advantages and Disadvantages of Flowcharts

II. DECISION TABLE STRUCTURE

This section is intended to introduce decision logic tables by describing their structure and the rules governing their use. Simply stated, a decision logic table is a tabular representation of all elements of a problem from conception to solution. Information displayed in this manner is easily comprehended even when the table of information represents a complex logical problem. The logic used in decision tables is similar to that which is used every day, with or without the aid of the computer.

A. THE ELEMENTS OF A TABLE

Before describing in detail the table itself, several definitions must be made clear. First, a decision rule is a statement that prescribes the set of conditions that must be satisfied in order that a series of actions be taken. For example, the following is a decision rule:

> If a laborer works more than 40 hours in a week, he must be paid his regular salary rate plus the product of his overtime hours times his hours in excess of 40.

The decision logic table is used to describe the possible decision rules derived above. That is, whether or not the laborer worked more than 40 hours in a given week.

The decision table itself is a structure for describing a set of related rules. Although other formats of decision tables exist, some of which are easier to use [18], they are all permutations of the basic format shown in Figure 2.

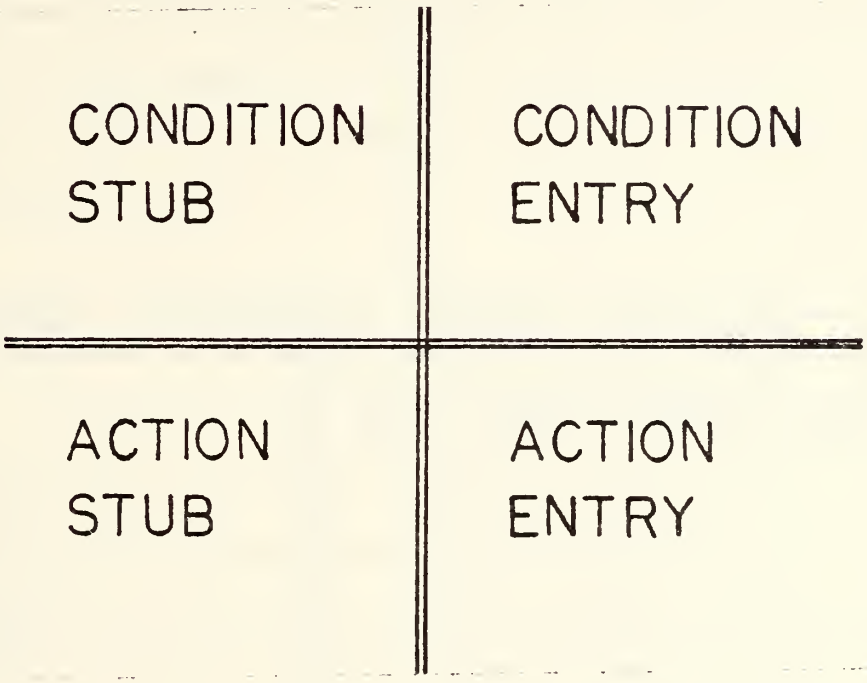


FIGURE 2. Decision Table Structure

As illustrated, the table is divided into four quadrants. The upper left quadrant, called the condition stub, contains all the conditions being considered for a particular decision rule. The condition entry, in the upper right quadrant, combines with the condition stub to form the condition that is to be tested.

In the lower left quadrant, is the action stub. It contains a simple statement of the actions resulting from from the conditions listed above the horizontal line. Action entries are displayed in the lower right quadrant. In this quadrant, the appropriate action resulting from the various combinations of responses to the conditions will be indicated.

As shown in Figure 3, the table also contains a section called a table header. Actually, this identifying data is required in order to distinguish it from all other tables in a given job.

TABLE HEADER RULE HEADER

	R1	R2	R3	R4
C1				
C2				
C3				
A1				
A2				
A3				

FIGURE 3. Basic Elements of a Decision Table

The information that might be found in a table header includes a table number, a table name, the table type, the number of rules, conditions and actions, and any other options locally established to simplify translation.

The various combinations of responses to conditions shown in the condition entry quadrant are called rules or paths. Each is given a number for identification purposes in the rule header portion of the table.

B. TABLE ENTRIES

If a condition in the condition stub is true, a Y is entered for that particular rule in the condition entry. Conversely, if the condition is false, an N would be entered. Where irrelevant situations occur, a "don't-care" is indicated by a dash (-).

Additionally, two other entries have been proposed to indicate mutual exclusion of one condition with another [18,8]. If the case arises within a single rule that the satisfaction of some test, indicated by a Y or N entry, makes some other entry a foregone conclusion, then the special entries '*' or '$' may be used to indicate this fact. The symbol '*' is used in place of an N entry under these circumstances, while the '$' is used in place of a Y entry.

The network algorithms and several sophisticated algorithms that attemot to minimize execution time of the translated table and/or provide completeness checking make excellent use of these implied truth values. Those algorithms provide completeness checking which accounts for the logically impossible rules introduced by condition dependency.

In the action entry quadrant, an X is entered to indicate that action which is to be executed for a particular rule. Any given action may be executed for any number of rules, however a rule may require more than one action and where an X is entered for each action in the action entry quadrant.

C. TYPES OF TABLES

There are three types of decision tables in current use. The limited entry table is the most popular and most often used [8]. Since the other two table types, extended entry and mixed entry, may always be transformed into limited entry tables, the preprocessor developed here allows only limited entry table input. The other two types of tables will be discussed, however.

1. Limited Entry Tables

The rules regarding the placement of information in each of the four quadrants of a limited entry table are

fixed and inflexible. The condition and its state must be restricted to the condition stub. The condition entry may only show the response Y (true), N (false), * (implicit N), $ (implicit Y) or '-' (don't care).

Likewise, specific actions must be fully identified within the action stub and permissible notations within the action entry sections are limited to an 'X' or a blank.

Table 3 shows a limited entry table in proper format. Note that entries prescribed for one quadrant may not extend into another and that every condition entry contains one and only one of the allowed symbols. Normally, limited entry tables are the best suited to computer applications [13].

LOAN TABLE	R1	R2	R3	R4
SATISFACTORY CREDIT LIMIT	Y	Y	N	N
FAVORABLE PAY EXPERIENCE	Y	N	Y	N
APPROVE LOAN	X		X	
REJECT LOAN		X		X

TABLE 3. Limited Entry Table

2. Extended Entry Tables

In extended entry tables, the variables to be tested are identified in the condition stub, while the condition entry must define the value or state of the variable. Likewise, in this type of table, the action stub names an action while the action entry will give the specifics for the action named.

As shown in table 3, the format is not quite as strict for this type of table. The use of this format may also tend to decrease the number of items in both the condition and action stubs.

LOAN TABLE	R1	R2	R3	R4
CREDIT LIMIT	OK	OK	TOO LOW	TOO LOW
PAY EXPERIENCE	OK	POOR	OK	POOR
LOAN	APPROVE	REJECT	APPROVE	REJECT

TABLE 4. Extended Entry Table

3. Mixed Entry Tables

The mixed entry table is a combination of the limited entry form and the extended entry form. Even though these two forms may be combined, one form must be used exclusively within each horizontal row of a table. Table 5 depicts the information from the previously used tables as a mixed entry table.

LOAN TABLE	R1	R2	R3	R4
CREDIT LIMIT	OK	OK	TOO LOW	TOO LOW
FAVORABLE PAY EXPERIENCE	Y	N	Y	N
APPROVE LOAN	X		X	
REJECT LOAN		X		X

TABLE 5. Mixed Entry Table

III. DECISION TABLE THEORY

The uses for decision tables vary greatly throughout the fields of business, science, and engineering. Whatever their purpose, a sound theoretical basis is needed to explore further the intricacies of their potential. This section is dedicated to fulfilling that need with a general overview of the background theory of decision logic tables and specific treatment of rule mask theories. This discussion is a prelude to the topics of table completeness and decision rule contradiction and redundancy.

A. GENERAL

As previously stated, a decision table is made up of a set of conditions, each of which may be evaluated as true or false at any given time. The truth or falsity of these conditions may be combined in various ways, along with a series of actions, to form a decision rule.

The series of actions contained in a particular decision rule are executed when a transaction is evaluated that matches the particular combination of truth or falsity of the conditions indicated by the particular rule.

The decision tables presented here are based on one of the Boolean algebra functions known as the AND function. It is considered to be the ordered set of Y's, N's, or dashes that appear as the condition entries for a particular decision rule. The application of the OR function can also be made in decision tables and it is described in some detail by Pollack, et al.[18].

In order to illustrate the AND function, the following table is presented with its associated AND functions.

	R1	R2	R3	R4
TYPE ≥ 60 WPM	Y	Y	Y	N
SHORTHAND ≥ 90	Y	Y	N	–
SALARY ≤ $5500	Y	N	–	–
HIRE	X			
DO NOT HIRE		X		X
REFER TO TYPING POOL			X	

AND function 1 = Y,Y,Y

AND function 2 = Y,Y,N

AND function 3 = Y,N,-

AND function 4 = N,-,-

FIGURE 4. AND Function Examples

Basically, to determine whether or not a decision rule is satisfied, evaluate the AND function for that rule, and check that it equals the required transaction. For example, the AND function of rule 3 would be satisfied if the job applicant could type 60 or more words per minute but could not take dictation at a speed greater than 90 words per minute. For this rule, condition 3, the possible salary, is of no consequence to the ultimate satisfaction of the rule.

B. CONDITIONS

So far, the word condition has been used numerous times without a complete definition. A condition is a variable factor affecting the action(s) to be taken in a given situation by its presence, absence, or change in value. Series of conditions with their associated rule entries make up, in part, decision logic tables. The symbol n will be used to represent the number of conditions, each denoted by "C_1", "C_2", etc., present in the table.

When a table is evaluated, the various conditions are found to be either true or false. This truth value is stored in a matrix M according to the following code proposed by Press[20].

$M_{i,1} = 1$ and $M_{i,2} = 0$ implies the condition is true

$M_{i,1} = 0$ and $M_{i,2} = 1$ implies the condition is false

Therefore, for N conditions the following matrix M would be formed:

$$\begin{bmatrix} M_{1,1} & M_{1,2} \\ M_{2,1} & M_{2,2} \\ M_{3,1} & M_{3,2} \\ \vdots & \vdots \\ M_{n,1} & M_{n,2} \end{bmatrix}$$

Each vector of the matrix thus formed is called a mask.

1. Structure of Conditions

Each condition is most often made up of two operands related by a relational operator. For input to the preprocessor developed here, the conditions must each be grammatically correct C language expressions. For instance, in its most basic form, one operand in a condition statement must be a variable, while the other may be a constant or variable. The relational operators may be any one of the following C language operators:

== <= >= < > !=

For example, consider the following three conditions in which a variable is compared to an integer:

C_1 : x < 10

C_2 : y >= 15

C_3 : z != 0

At the time of evaluation, the truth value is determined for each C_i. Given that x = 5, y = 20, and z = 0, the following matrix containing two masks would be obtained:

$$\begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Condition statements may also be made up of subroutine calls or variables or even any combination of these separated by logical operators. They must, however, evaluate as logically true or false.

2. Condition Dependency

Between any two pairs of conditions, there exists either dependence or independence. Basically, two conditions are dependent if they both have the same condition variable as an operand. Conversely, two conditions are independent if there is no common condition variable used as an operand.

There are two types of dependence. First, there is mutual exclusion dependency. This case occurs when for any pair of conditions C_i and C_j, there is no value of the common condition variable such that their mask entries are both equal "1,0", true. However, this is not to say that both conditions may not be false at the same time.

By extention to more than two conditions, it may be said that any number of conditions are mutually exclusive if at any point in time every two conditions in each of the pairs of conditions are mutually exclusive.

The second type of dependency is termed overlapping dependency. Overlapping dependency occurs when there can exist at least one value of a condition variable common to a pair of conditions such that both conditions are true. Other combinations of truth and falsity may also occur. Condition dependency thus dictates that certain combinations of condition values are impossible events. These impossible events represent impossible rules and need not be considered by the programmer when describing the program logic. However, machine checking for condition dependency is seldom implemented in translation algorithms. This causes the machine to interpret the decision logic table as being incomplete.

3. Condition Entry Notation

The condition entry portion of a decision table contains one of the following entries for a condition C_i, which have the meaning given:

'Y' => C_i is required to be true.

'N' => C_i is required to be false.

'-' => C_i is immaterial.

'*' => C_i is false, if some other explicit condition is proven.

'$' => C_i is true, if some other explicit condition is proven.

In the arguments to follow, the dash or '-' will be denoted by I_i for a condition C_i, where the condition is immaterial and C_i need not be proven either true or false. Also, it should be pointed out that

$$I_i = Y_i + N_i$$

where + is an inclusive OR

Analogously, the symbols '*' and '$' indicate that the condition they represent need not be proven false or

true. They implicitly represent both Y_i and N_i for a condition C_i and thus have the full power of both, if tested.

C. AND FUNCTIONS

In the discussion to follow, an AND function, B_j, will be considered to be defined as:

$$B_j = W_{1,j} \& W_{2,j} \& W_{3,j} \& \ldots\ldots \& W_{n,j}$$

Where W_i is a vector representing any one of the possible condition entries for condition i and '&' is the Boolean operator AND.

Each independent condition may require W_i to be Y_i, N_i, or I_i. Dependent conditions may take on the implicit requirements $\star_i$ and $\$_i$, but these are special cases of Y_i and N_i, respectively. Therefore W_i may be expressed in one of three states: Y_i, N_i or I_i. Thus, the number of possible forms of an AND function is 3^n.

Recall that the matrix M represents the truth or falsity of n conditions. Given a particular matrix M, it may be determined for M whether $V(B_j)$, the logical value of B_j, equals 1 or 0 by first making appropriate entries for each $W_{i,j}$ of B_j, according to the rules indicated in Figure 5. For example, suppose

$$B_j = (Y\ Y\ N\ \star\ I\ Y)$$

and

$$M = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Then, according to the replacements indicated in Figure 5, we have

$$B_j = (1\ 0\ 1\ 1\ 1\ 1)$$

Therefore, $V(B_j) = 0$. Had the resulting B_j contained all 1's, the logical value, $V(B_j)$, would have been 1.

$W_{k,j}$	$V(C_k)$	Replace with
Y	0 1	0
Y	1 0	1
N	0 1	1
N	1 0	0
*	0 1	1
*	1 0	1
$	0 1	1
$	1 0	1
I	0 1	1
I	1 0	1

FIGURE 5. Table of Replacements for Determining $V(B_j)$

As an example, consider again the following conditions:

$$C_1 : x < 10 \qquad C_2 : y >= 15 \qquad C_3 : z != 0$$

Given that x = 5, y = 20, and z = 0, the following matrix was obtained:

$$\begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

From a typical decision table that may be easily formed, the following AND function is present:

$$B_1 = (Y\ Y\ Y)$$

Again, according to the replacements indicated in Figure 5, we obtain

$$B_1 = (1\ 1\ 0)$$

And $V(B_1) = 0$. The first AND function tested is not satisfied and its associated action(s) will not be done. Another function must be considered.

1. Dependency of AND Functions

Dependency among AND functions is somewhat different than that among conditions. Two AND functions, B_i and B_j, are dependent if for at least one set of values of the conditions variables and requirements, both $V(B_i) = 1$ and

$V(B_j) = 1$. Otherwise, the AND functions are independent and no one set of the condition variables set $V(B_i)$ and $V(B_j)$ to 1.

For example, consider the following two AND functions:

$$B_3 = Y , Y , Y , N$$

$$B_4 = Y , Y , Y , N$$

Then for a set of values of the condition variables that yields

$$M = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

both $V(B_3) = 1$ and $V(B_4) = 1$. Thus, B_3 and B_4 are dependent. Had either $V(B_3) = 0$ or $V(B_4) = 0$, then B_3 and B_4 would have been found to be independent.

2. Definitions

A pure AND function is one that contains no "I" entry. For example,

$$P = Y , * , \$$$

is a pure AND function.

A decision rule is simple if it contains a pure AND function. A mixed AND function is one that contains one or more I's. A decision rule is complex (or compound) if it contains a mixed AND function.

D. THEOREMS FOR AND FUNCTIONS

In the theorems that follow, a table, T, is assumed to comprise all AND functions that can generate from the conditions of that table. The theorems are presented for informational purposes only. Detailed proofs are presented by Pollack [18].

THEOREM I. Within table T, two AND functions are independent if, in at least one position, one function contains a Y and the other function contains a N. Otherwise, they are dependent.

For an illustration of theorem I, consider the following incomplete table.

	AF1	AF2	AF3	
C1	Y	—	Y	
C2	—	N	Y	
C3	—	—	N	

AF1 and AF2 are dependent since there does not exist at least one Y,N pair for the three conditions.

AF2 and AF3 are independent because a Y,N pair exists for condition C_2.

THEOREM II. Within table T, each pure AND function is independent of every other pure AND function.

Consider the following table for an illustration of theorem II.

	AF1	AF2	AF3	AF4
C1	Y	Y	N	N
C2	Y	N	Y	N

By definition, all four AND functions are pure. From theorem I, they are independent. In this example, there are no other pure AND functions possible and therefore each pure AND function is independent of every other pure AND function.

THEOREM III. Within table T, every mixed AND function that contains I in positions ($1 <= r < n$) is dependent on each of 2^r pure AND functions of T.

Consider the following partial table.

	AF1	AF2	
C1	Y	Y	
C2	N	Y	
C3	—	Y	

AF1 expands to contain the following pure AND functions.

AF1a	AF1b
Y	Y
N	N
Y	N

Referring to theorem I, AF1 is dependent on AF1a and AF1b.

THEOREM IV. Table T, based on n conditions, contains one, and only one, set of 2^n independent pure functions.

As an illustration of theorem IV, consider the following table.

	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8
C1	Y	Y	Y	Y	N	N	N	N
C2	Y	Y	N	N	Y	Y	N	N
C3	Y	N	Y	N	Y	N	Y	N

With the total number of conditions equal to 3, the total number of pure AND functions should be $2^3 = 8$, according to theorem IV. As can be seen above, no other pure AND function exists that does not duplicate one of those in the table.

THEOREM V. Any decision rule that contains I in r positions (0 <= r < n) of its AND function is equivalent to 2^r simple decision rules.

Theorem V is illustrated by the following partially complete table.

	R1	R2	
C1	Y	N	
C2	—	N	
A1	X		
A2		X	

The complex decision rule, R1, is equivalent to $2^1 = 2$ simple decision rules, R1a and R1b, both of which contain pure decision rules.

R1a	R1b
Y	Y
Y	N
X	X

IV. PREPARING DECISION TABLES

A. BASIC TECHNIQUES

This section presents two methods for the development of decision tables. The classical technique and the progressive rule development technique, both of which follow the formal preparation rules presented in an earlier section.

In the classical technique, all possible combinations of conditions are considered and a matrix is produced in the condition entry which represents all possible simple rules. Next, the table is simplified by repeatedly combining several simple rules into a complex rule, thereby producing fewer rules. The process becomes almost mechanical, however, and it is possible to lose sight of the total problem logic.

The other method, by progressive rule development, is based on formalized rules adapted from flowcharting. The logic of the problem is considered step by step and the table prepared as the problem is studied. Normally, this method is somewhat faster than the classical one. Because the development of the table is in step with the logical analysis of the problem, the total logic can always be seen.

It should be pointed out here that the form of the decision table should reflect its ultimate use. When preparing a decision table for preprocessing, compact, sophisticated logic should be reflected in the table. Alternatively, if a table is to be used purely for documentation purposes, it would be best for the table to be laid out in a simple, easily readable form. The fact that oversophistication in compressing logic and in minimizing the number of rules produces a table which is more difficult for the ultimate user to understand should be kept in mind. Furthermore, if care is not used, errors may inadvertantly be introduced.

B. CLASSICAL TECHNIQUE

The classical technique emphasizes the development of a matrix representing all the simple rules for the given conditions. Then the full matrix will be reduced, if possible, to a fewer number of rules by combining the simple rules to form complex rules. The following seven steps may be followed almost mechanically to produce a logically correct and relatively concise table.

1. List all conditions

2. List all actions

3. Complete condition entry (for full matrix)

4. Complete action entry

5. Consolidate and form complex rules

6. Check table

7. Transcribe final version and recheck

This technique can be used for all three types of tables (limited, extended, and mixed entry). Each of the seven steps will be described in the following sub-sections.

1. List Conditions

By thorough study of the problem under consideration, all the relevant conditions may be determined and listed. At this point, the statement of the condition should be as clear and concise as possible. In order to avoid logically correct, but complicated statements, negative expressions should be avoided whenever possible. A negative statement relies on a double negative for the positive case. For example, the condition "no space available" gives Y (out of space) and N (space available).

As a general rule, conditions which are not independent should also be avoided. Often, where one condition includes another, there may be some misunderstanding of the problem at hand. Conditions which are not independent produce impossible rules and incomplete logic tables.

Logically, the sequence of the conditions tested does not affect the validity of the table, but it does affect the ease of reading and table construction. The

basic guide to follow is to list conditions in order of most likely satisfaction. When their relative likelihood is not known, listing in the sequence in which identified is a good starting point. It is imperative to list all possible conditions before proceeding to the next step.

2. List Actions

Listing the actions next allows a double check to ensure that all the conditions have been listed. Action statements are generally easier to formulate than condition statements and are generally given as some sort of command. For convenience, actions should be listed in the sequence in which they are to be performed. This rule is mandatory when advanced techniques, such as recursion or table linkage, are utilized.

3. Complete Condition Entry

At this step, the condition entry part of the table is filled in. The object here is to state all the rules which represent all combinations of conditions with no repetition or omission of any combination. As previously stated, for a limited entry table there are 2^n simple rules, where n is the number of conditions.

At the conclusion of this step, a completed matrix is formed in the condition entry portion of the table consisting of all the possible simple rules. If the above

procedure is followed, each rule will be unique and all rules will contain every combination of conditions.

4. Complete Action Entry

At this point, each rule is examined. The condition entry is considered and the appropriate actions indicated. Any action required must be consistent with any other action required. In the event contradictions among actions can be identified at this point, they must be resolved by specifying an appropriate error action. Above all, it is vital to be consistent in handling any ambiguous combination of conditions.

5. Consolidation

In consolidating a limited entry table, consider two rules with identical actions at a time. For each pair, the two rules may be consolidated if all the condition values are the same except one pair. For the condition with the unmatched pair, a dash is entered. This one complex rule then replaces the two simple rules. Continuing in this manner will result in a smaller, more manageable table.

6. Check Table

At each of the stages previously described, the work done on the table should be checked. The earlier an error is detected, the easier it is to rectify. The checks that

should be applied fall into two broad categories: checking for content and checking for structure.

Checking the content should ensure that the action entries associated with each simple rule on the unconsolidated table are correct. Checking the structure of the final table is an attempt to ensure that the table contains no contradictions or redundancies.

7. Make Final Version

Once a table has been checked, it may be necessary to transcribe it to produce a final version which can then be used. The condition and action stubs should be checked to make sure that they are clear.

Normally, the conditions are listed so that those with the most "don't care" entries are at the bottom of the list. Similarly, the sequence of rules can be altered so that those rules containing the most "don't care" entries appear first. Large tables may be divided into smaller portions for checking.

C. PROGRESSIVE RULE DEVELOPMENT TECHNIQUE

Progressive rule development is based on standard techniques for preparing flowcharts. Whereas the classical technique requires that all possible combinations of conditions be defined, progressive rule development requires that conditions be written on the table as they are identified.

Each rule, including the action entry, is entered as the problem is analyzed.

The procedure for progressive rule development as proposed by London [11] is enumerated below. Note that the steps are repeated until the complete table has been formed.

When all possible conditions have been considered, the table should be checked for contradictions and redundancies. The following sub-sections point out the major points to be kept in mind at every step.

1. Consider a Condition

At this point, a condition should be clearly entered into the condition stub. As a starting point, enter a Y (true) response in the condition entry adjacent to the condition.

2. Consider Further Conditions

Determine what other conditions are necessary before action can be taken. They must also be entered in the table as in step 1. The action portion of the table may then be completed for this newly formed rule.

3. Form Next Rule

The next rule may be formed by transcribing the previous rule, changing the last condition entry for which all values have not been considered. For example, the last Y

value entered should be changed to an N. All values above the changed value are kept the same.

D. AN EXAMPLE

This section presents a solution to the following sample problem using the classical technique.

Hiring a Receptionist

> A new receptionist is needed for an insurance company. She must be able to type at least 60 words per minute and take dictation at a minimum of 90 words per minute. All applicants should be willing to work for a salary not greater than $5500 a year. All applicants who meet typing requirements but not the dictation requirements will be referred to the typing pool.

The first step is to identify the conditions that must be met. They are then placed in the condition stub of the decision logic table in some order of precedence (see Table 6). All possible actions should be placed in the action stub next.

	R1	R2	R3	R4	R5	R6	R7	R8
TYPE ≥ 60 WPM	Y	Y	Y	Y	N	N	N	N
SALARY ≤ $5500	Y	Y	N	N	Y	Y	N	N
SHORTHAND ≥ 90	Y	N	Y	N	Y	N	Y	N
HIRE	X							
DO NOT HIRE			X		X	X	X	X
REFER TO TYPING POOL		X		X				

TABLE 6. Hiring a Receptionist

The number of rules required to consider all possible combinations of conditions is:

$$\text{Number of rules} = 2^n$$

where n = the number of conditions

In this case, 8 rules are required, as indicated in Table 6. Note, however, that rules 5 thru 8 may be combined due to the fact that failing the typing condition results in the action "Do Not Hire" in all cases. Thus, it can be seen that the table may be dramatically simplified immediately.

Further, the table may be reduced to the one depicted in Table 7 by noting the fact that upon satisfying condition 1 but failure of condition 2, an ability to take shorthand at a rate of 90 or more words per minute, results in that

applicant being refered to the typing pool. Note that rearranging conditions in order of least number of don't care entries results in the final, bifurcated form shown. That is, it is arranged whereby each condition has its Y answers grouped together and its N answers grouped together to form paths.

	R1	R2	R3	R4
TYPE ⩾ 60 WPM	Y	Y	Y	N
SHORTHAND ⩾ 90	Y	Y	N	–
SALARY ⩽ $5500	Y	N	–	–
HIRE	X			
DO NOT HIRE		X		X
REFER TO TYPING POOL			X	

TABLE 6. Hiring a Receptionist - Final Solution

V. PREPROCESSOR DESCRIPTION

A. THE ALGORITHM

There were a multitude of different, sometimes opposing, attributes that the desired algorithm was to possess. These ranged from the traditional considerations of output module size and execution speed to restrictions arising from the intended implementation computer facility.

The choice of compiler, interpreter, or preprocessor was resolved in favor of the preprocessor due to the considerations dictated by the general-purpose, multi-user, interactive operating system UNIX [7], as implemented at the Naval Postgraduate School, and its support of the programming language C [23]. Preparing either a compiler or interpreter would have entailed duplication of that support to some degree.

Algorithms have been developed to attempt to minimize execution time, execution module size, or both [19]. The minimization effort arose from the consideration that the prepared execution module was to be used repeatedly, with the preprocessor itself being used relatively infrequently on individual tables. However, in the academic situation, the emphasis has been placed on preparing a working module rather than preparing a production type module. This

indicates that the preprocessor itself will be used rather frequently and the output module will be used very seldom. This led to the realization that the preprocessor size and execution time were of greater importance than output module size and execution time. It was therefore considered desirable for the preprocessor code to be small in size and quick in execution. Further, the data area of individual users was to be relatively small yet still capable of handling more than ten to twelve conditions in the decision logic table.

The final attribute of major importance was that the algorithm be capable of being implemented with a minimum of user skill. It was felt that several sophisticated algorithms [18,22,28,24], while of major importance in both the academic and industrial communities, demanded too much user input to be desirable for beginning decision logic translator users.

The various network algorithms described [18,19,28,10] were eliminated as a class since the data area available in our mini-computer was insufficient for ten to twelve conditions and the preprocessor execution time was estimated to be excessive.

Rule mask algorithms have been shown to be highly efficient with respect to storage requirements (execution module size) [19], translator size, and execution time, but poor with respect to execution module run time since each

condition had to be tested to prepare the mask and then this mask compared with all the rule masks. This objection faded when it was recognized that the target user group would, in general, be but slightly concerned with performing the statistical background work necessary to provide the input data to obtain truly optimal execution time code.

The rule mask technique of Press [20] has been shown to be very good with respect to execution module run time [19]. Additionally, the target user group was expected to be capable of programming decision logic tables using the input required by this algorithm with relative ease.

For these reasons, the choice of an algorithm was that of Press [20]. This algorithm built a rule mask for each rule. Code was generated to sequentially evaluate each condition and construct a test mask from the results. The rule masks were then scanned to find one that matched this test mask.

This algorithm, on one hand, did not require a large data area, which would have been the case with a network algorithm. Yet, on the other hand, the programmer was given nothing to control execution time of the output program other than simply placing the rules in decreasing order of expected frequency of satisfaction.

In order to provide a smaller preprocessor module to document the grammar of the decision logic table, and to

simplify any future changes to that grammar, YACC, a compiler-compiler developed by Bell Laboratories [6], was used in the construction of DELTRANS, the decision logic translator proposed here.

B. GENERAL OPERATION

As previously stated, DELTRANS was designed to be a sequential testing rule mask decision logic table translator. This meant that for each decision logic table DELTRANS was to prepare a rule mask to match each rule, generate code to test each condition sequentially and set a test mask, and finally generate code to test the test mask against the rule masks, searching for a match.

To accomplish the enumerated tasks DELTRANS was conceived to operate in five distinct but interdependent phases. The five phases were designated copy, data area initialization, data input, computation, and finally code generation. When more than one table was to be preprocessed, DELTRANS returned to the copy phase.

As designed, DELTRANS began execution in the copy phase. Code was merely transferred from the input to the output file, removing comments while searching for the first up arrow (↑) not within quotes, which indicated the start of a decision logic table.

At this point DELTRANS entered the data area initialization phase. In this phase, DELTRANS read a list of user options such as the number of actions or conditions, and initialized the internal structures in preparation for table input. The user was provided with a great deal of flexibility in both size and format, and considerable error checking was performed during this phase.

If all initialization input was in order, the preprocessor proceeded to the third phase, data input. In this phase, the table was read and its contents sorted and stored for the next phases. Once again, extensive error checking was done during this phase.

When the final up arrow in a table was read and the data input phase complete, DELTRANS shifted into the computation phase. In this phase there were two major events, ambiguity and completeness checking and the construction of the individual rule masks.

The final phase of code generation was the point of generation of both the output code and a formatted decision logic table for use in debugging. DELTRANS returned to the copy phase to pass on any additional code or prepare for a following table.

All but the final phase of the preprocessing could cause fatal errors. If an unexpected end-of-file was encountered, an error message resulted and the output buffers were

flushed into the output file. Otherwise, the standard recovery technique was to search for the up arrow, which was assumed to mark the end of the current table, and then restart from the copy phase.

C. AMBIGUITY AND COMPLETENESS CHECKING

DELTRANS was designed to perform two distinct types of table logic checking for the user, but had no capability of correcting any errors exposed during this logic checking. Completeness checking was attempted only after the ambiguity checking was completed and no errors found.

Ambiguity checking, as performed by DELTRANS, was based on two fundamental requirements for all decision logic tables [19]. First, every rule must have at least one associated action. And second, each distinct combination of truth values for the given set of conditions must satisfy exactly one rule.

The first requirement arises because for every set of conditions some action is, in fact, intended or should be. Whether that action be return to the calling point in the program, halt program execution immediately, or enter an infinite no-operation loop, some program action is intended.

Checking for redundancy and inconsistency in table construction has been implemented by comparing the rules to insure that between each pair of rules there exists at least

one condition row with opposing logic entries for the two rules. If no opposing logic entries are found, their identical actions indicate a redundant table and differing actions indicate an inconsistent table.

Examples of both cases can be readily observed in table 8. Rules R1 and R2 are redundant while R3 and R4 are inconsistent. Note that the implicit entries are considered to be equivalent to the explicit entries and therefore there is a lack of opposing logic entries.

	R1	R2	R3	R4
C1	Y	$	N	N
C2	$	–	N	*
C3	–	N	–	Y
A1	X	X		X
A2			X	

TABLE 8. Redundancy and Inconsistency Example

Only if the input decision logic table was non-ambiguous did DELTRANS attempt to determine if that table was complete. Completeness testing on an ambiguous table is unnecessary. As previously noted, Pollack, Hicks, and Harrison [18] have proven that each decision logic table

contains 2^n independent simple rules, where n is the number of conditions. They have also proven that all rules represent 2^n simple rules, where n is the number of "don't cares" in that rule. DELTRANS has incorporated these two theorems in its completeness testing.

If a decision logic table contains two or more dependent rules it is said to be ambiguous. In that case the two theorems on completeness would not apply. Therefore the ambiguity checking was designed to preceed the completeness checking.

When checking for completeness, the preprocessor was designed to scan each rule for a count of the "don't" care" entries in that rule. For each rule, 2 was raised to the power of this count and the resulting value added to a tally sum for the entire table. When all rules had been scanned, this tally was compared with the value of 2 raised to the number of conditions. If these two were equal the table was complete; otherwise the table was incomplete.

If an input decision logic table was found to be complete, the else/error rule can never be satisfied and is therefore superfluous. Since, by design, DELTRANS required an else/error action, it was necessary to provide the programmer with the capability to input a null action (not a no-operation) to be used with complete tables.

If the count of simple rules in a decision logic table revealed that not all possible rules had been enumerated, the else/error action was examined. If that action was a null action, then DELTRANS was designed to output a warning message indicating that an incomplete table had been encountered. Of course, if the program had specified a valid else/error action then the decision logic table is, by default, complete.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

Throughout the available literature, decision logic table structure and terminology was found to be rather straightforward and standardized, as presented here. This has facilitated the programmer's use and understanding of decision logic tables.

The theory upon which decision logic table construction and translation has been based was found to vary between extremes. Pollack and others [18] have presented clear and direct foundations for construction and translation. Some algorithms were discovered which were based more on intuition than theory [28], while others were founded in theory so complex that programmers have had difficulty in grasping the logic of a given problem [4,22,24].

The advantages that decision logic tables offer have shown that every programmer should at least be introduced to decision logic tables and thus be able to use this powerful tool.

Decision logic tables can be a powerful aid in effective communicating, both man-to-man and man-to-machine, in programming, and in documenting. The format of decision logic

tables permits organization and concise visual presentation of complex logic. Decision logic tables also provide the programmer with a very effective tool for insuring that the program logic is both correct and complete, items that other methods tend to obscure.

Additionally, since decision logic tables are both easy to construct and modify, and may be used as computer input, decision logic tables, when properly used, can be an extremely effective tool for communicating, programming, and documenting.

Of the many decision logic table translators available[14], DELTRANS, as proposed here, was the only known available translator designed for implementation on the UNIX timesharing system for the C programming language.

The ultimate value of DELTRANS lies in its versatility of application throughout management, scientific, and engineering fields. Decision logic tables themselves provide a simple method for recording logic so that all elements of a decision are precisely defined. Tables make it possible for managers, scientists, and engineers to use computers directly. Much subsequent programming and coding may be eliminated.

DELTRANS, as developed, fills that gap between a C programmer with decision tables and the C language compiler. The Naval Postgraduate School has been provided with a tool

for use in introducing students to the use of decision logic tables. A tool that until now has not been available.

B. RECOMMENDATIONS

Several refinements to DELTRANS have been suggested to further enhance its utility. Prior to enumerating the most important of these refinements it should be pointed out that each of these requirements conflicts with some of the design criteria used in constructing DELTRANS.

Additional completeness testing and error checking capabilities would assist the programmer with complex logic. If the necessary space and time were deemed appropriate, the preprocessor could be so modified to take full advantage of the implicit entries during completeness checking. Further coding could provide for automatic error correction of a number of programming errors, for example combining redundant rules.

An alternate conversion algorithm could be implemented. By using one of the network algorithms that has been proven to provide minimum execution time output, the capabilities of DELTRANS would be enhanced. Since the data structures for holding the actions, conditions, and rules and for linking the rules to the actions were built and maintained by DELTRANS, the implementation of an additional coding algorithm would be simplified. However, these algorithms would

require additional programmer input and additional time and space for the preprocessor.

If deemed appropriate, the required increase in size and decrease in speed in the preprocessor could be accepted and DELTRANS could be modified to accept extended and/or mixed entry conditions.

A conversational translator could be developed using DELTRANS as a base. This would greatly reduce the file manipulating required of the programmer under DELTRANS.

A final recommendation is that decision logic tables be presented to students as a part of an introductory computer science course. Even if tables must be hand translated, decision logic tables can provide great assistance to the programmer, whether he be a beginner or experienced.

APPENDIX A - DELTRANS USER'S MANUAL

Effective Use of DELTRANS

DELTRANS is designed to be utilized by those fluent in the design and construction of decision logic tables. Those without such a background are directed to the appropriate sections of the parent thesis itself and Solomon Pollack's work Decision Tables: Theory and Practice [18] for an introduction to decision logic tables.

Additionally, some basic familiarity with the UNIX operating system is assumed; specifically, using the editor, programming in C, and file manipulation. The paper "UNIX for Beginners" by Kernighan [7] is an excellent starting point.

Only with a thorough grasp of the concepts of both decision tables and UNIX may a user of DELTRANS reap its intended benefits as described herein.

DELTRANS USER'S MANUAL

CONTENTS

I. INTRODUCTION

Decision logic tables describe decision rules. A decision rule consists of a set of conditions plus a set of actions. The relationship between conditions and actions is of the IF-THEN type. More specifically, if the given conditions are met, then the corresponding actions are taken.

DELTRANS allows C programmers to convert a C program containing one or more decision logic tables into a C source program ready for compilation.

The user needs only a basic knowledge of the C language in order to use DELTRANS. The decision table input, nested within a C program, must conform to the specifications of a C-oriented decision table language presented in section II of this manual. The language described combines decision table capabilities with many of the features of C.

DELTRANS processes one decision table at a time. It reads, decodes, and edits each line of the table. Messages are printed on the terminal when errors are detected. The decision table is output as C source code on a specified file. Optionally, a formatted listing may be obtained.

The decision logic translator is designed for use on the PDP-11 with the UNIX operating system and 64K bytes of user program storage.

A. PURPOSE

The purpose of DELTRANS is to provide an alternative to the C programmer when faced with a complex logical situation. The preprocessor is designed to be convenient for preparing C programs containing decision logic tables at a remote terminal. A decision logic table may be included within any C program, along with any syntactically correct C expressions.

B. FUNCTIONS PERFORMED

Briefly, DELTRANS opens and reads from the input file specified by the user and preprocesses the C program contained therein. The default is input from the terminal itself. Source code is placed on the output file also specified by the user. Its default name is 'd.tab.c'. This file is initially opened and then closed, along with the input file, when preprocessing is complete. Each decision logic table is coded as a separate subroutine in this file.

Optionally, a formatted and thus very readable copy of each decision table contained in the input file may be

redirected to the line printer or any other file upon completion of preprocessing.

Additionally, the decision logic table is examined for ambiguous or incomplete logic.

C. LIMITATIONS

The processor limitations described here are due to the dimensions of various arrays internal to the preprocessor. Most of these limitations may be relaxed by minor alterations to the preprocessor; a consultant may be of some assistance in this endeavor.

The following list of maximum values must be strictly adhered to avoid processing errors.

Maximum number of conditions = 16

Maximum number of actions = 32

Maximum number of rules = 64

Maximum stub length = 31 characters

Additionally, the name of the table subroutine may be no more than 8 characters.

The preprocessor was written using a rule mask scanning technique. A fundamental assumption required when using

this technique is that every condition is tested when preparing a mask which is, in turn, used to scan for the proper rule. This fact forces the programming limitation that each condition return a valid test regardless of the results of previous tests. Note that the conditions are tested sequentially, first to last.

Note especially the reserved words dda, ddb, and ddc. These may not be used in the logic in decision logic tables since the preprocessor uses these as names for the masks.

Finally, the up arrow, except when inside quotes (single or double) will be read as a definite signal to the preprocessor...beware!

D. ADDITIONAL BACKGROUND

The preprocessor was written during January and February, 1977, as a thesis project by Lt. J.F. Keller and Capt. R.W. Roesch.

Subroutines required by the system are as follows:

getc putc fopen printf

exit fflush fcreat

The YACC library routines are also required.

II. GENERAL INFORMATION

A. ACCEPTABLE INPUT

A decision logic table may occur anywhere within a C program; however, it is normally considered as a procedure and should be treated accordingly. As shown in Figure 1, the table itself is first recognized by the preprocessor by the occurrence of a left arrow ('←') as the first character of any line within a C program.

The table itself is divided into three distinct sections, separated by the up arrow ('↑'), as depicted in Figure 1. As shown, the sections are identified as the option section, the condition section, and the action section.

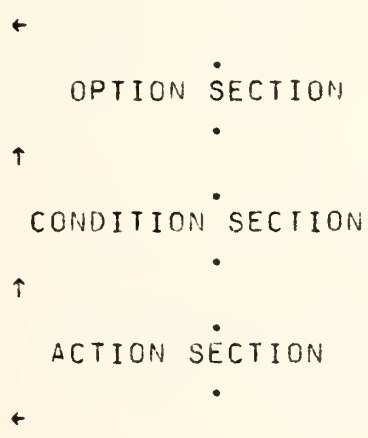


FIGURE 1. Decision Table Format

```
 1 int r[6]     { 0, 1,-2,-3, 2,-3};
 2 int s[6]     {-4, 9,-7, 0,-4,-1};
 3 int t[6]     {-1, 3,-2,-1, 0, 3};
 4
 5 main () {
 6     int k;
 7     for (k=0; k<6; k++) {
 8         printf("\nThe numbers to test are :\n");
 9         printf("\tR = %d\n\tS = %d\n",r[k],s[k]);
10         printf("\tT = %d\n",t[k]);
11         logtab(k);
12     }
13 }
14
15 rpos () {
16     printf("R is positive.\n");
17 }
18
19 ←
20 n logtab (c)     int c;  {
21               // this is a comment
22 r 5 c 3
23 a 4
24 e @
25 d    int a;
26     char b;
27 ↑
28 r[c] < 0 @ y(1 - 3 , 4 ) N(5);
29 s[c] < 0 @ y(1,2)        n(3 ,4);
30 t[c] < 0 @ y(1-4) -(5)
31                   n(2-3);
32 ↑
33 printf("T < 0 : ") @ 4,1;
34 printf("S < 0 : ") @ 1,2;
35 printf("R < 0 \n ") @ 1,2,3,4;
36 rpos()          @ 5 ;
37 ←
```

FIGURE 2. Example Program with a Decision Logic Table

Each section has a specific format and although there is much freedom of input allowed, several rules must be followed as described in the following sub-sections.

1. The Option Section

The format of the option section is relatively free, however several items must appear and be initialized. Documenting comments follow the rules of C and thus may appear anywhere within this section. In accordance with those rules, they must be preceeded by a double slash ('//') or preceeded by '/*' and followed by '*/'.

The possible options are listed below.

1. 'a' or 'A' : number of actions : required
2. 'c' or 'C' : number of conditions : required
3. 'd' or 'D' : declarations : optional-see below
4. 'e' or 'E' : else / error action : required
5. 'n' or 'N' : subroutine name : required
6. 'r' or 'R' : number of rules : required

If the declaration option ('d' or 'D') is used in a table, it must be the last option used in the option section of that table. All variables local to the decision table itself must be declared to avoid future compilation errors. As shown in Figure 2, the declarations must be syntactically

correct C declarations. DELTRANS implements this feature by passing untouched to the output file all code between the 'd' (or 'D') and the up arrow at the end of the option section.

Additionally, the following options must be specified in the format indicated:

a. The number of conditions:

c <number> or C <number>

b. The number of rules:

r <number> or R <number>

c. The number of actions:

a <number> or A <number>

Again, as shown in Figure 2, these options may be specified in free form with one or more spaces between a letter and a number.

Another required option is the name or 'n' option. It must also appear before the declarations and must be of the following format:

n <name> (<formal parameters>) <type specifications> ; {

where

'n' may be 'N'.

<name> may be from 1 to 8 characters

<formal parameters> is optional depending upon required parameters.

<type specifications> are required for all parameters declared.

For example, line 20 in Figure 2 names the output subroutine "logtab" with one parameter, "c". Compare with the output code in Figure 4.

The final required entry in the option section is the else/error action. The preprocessor will physically code this as the final "else" action is the output code. The format of the else/error is an 'e' (or 'E') followed by any number (including zero) of blanks and tabs followed by a string of at most 31 characters followed by '@'. The 31 characters must form a valid C expression or group of expressions.

2. The Condition Section

The condition section is organized as a series of condition lines with the end of the section indicated by an up arrow.

Each condition line contains four entries: a condition stub of up to 31 characters; an at sign ('@'), which

indicates the end of the condition stub; an optional list of truth values corresponding to numbered rules; and finally a semicolon, which indicates the end of the condition line.

The condition stub is copied character for character (maximum 31), up to the at sign, directly into the appropriate internal structure.

The format for the list of truth values is relatively free-form in that tabs, spaces, and newlines are ignored when appropriate. The default entry for each rule is a "don't care" or dash. To overwrite with any logic letter (Y,y,N,n,*,$,-), that logic letter is placed in front of a set of parentheses containing the list of rules for which that logic letter is to be entered.

This list of rule numbers must contain at least one rule number and may be in either or both of two formats. The first format is a list of single rule numbers, separated by commas, in any desired order. The second or "through" format uses one rule number, a dash, and then another rule number. The effect of this format is to enter the selected logic letter into each rule starting with the first and including all the rules through the last. See Figure 2 lines 28-31 for examples of condition lines.

When all the logic letter overlays have been entered, a semicolon is entered to alert DELTRANS that the

end of the truth values for the current condition has been found and that the preprocessor must proceed to either accept the next condition stub, or, if the up arrow is found next, proceed to decode the action section.

DELTRANS was constructed with the intention that it be used as a basis for additional work in providing decision logic table processing capability. The original implementation does not use the implicit entries ('*' and '$') in either completeness testing or in construction of the output code. The programmer is nonetheless strongly encouraged to use these entries since they help to bring out the logic of the problem at hand.

3. The Action Section

The action section is organized as a series of "action lines" with the end of the section indicated by a left arrow. The left arrow is used since the end of the action section is also the end of the input for the current table.

Each action line contains four entries: an action stub of up to 31 characters; an at sign to indicate the end of the action stub; a list of rules for which the action is to be performed; and a semicolon to indicate the end of the action line.

The action stub is copied character for character (maximum 31) up to the at sign directly into the appropriate internal structure.

The format of the list of rules is relatively free-form in that, once again, tabs, spaces, and newlines are ignored when appropriate. The default entry for each rule is not to perform the current action. Thus, to indicate the action is required for a list of rules, that list of rules, separated by commas, is entered. No particular order of rule numbers is required. See Figure 2 lines 33-37 for examples of action lines.

When the list of rules has been entered, a semicolon is input to alert DELTRANS that the end of the current action line has been discovered and that DELTRANS must perform the actions required to determine whether or not to terminate this phase of operations.

B. PROGRAMMING TECHNIQUES

Although any beginning programmer familiar with C, UNIX, and decision logic tables should be able to construct a valid input decision logic table for DELTRANS, there are three specific aspects of input programming that a more advanced programmer should know.

1. Rule Sequence

One process by which a programmer may increase the average execution speed of the output execution module is by proper ordering of the rules in the decision logic table. After testing the various conditions to prepare the test mask, that test mask is compared with each rule in sequence. Thus, if the programmer will code the input decision logic table so that the rules will be listed in decreasing frequency of satisfaction, the average execution time of the output execution module will be decreased.

2. ELSE / ERROR Action

The else/error rule and its associated action should be the subject of considerable thought when programming decision logic tables. If they are improperly used the execution speed of the ouput module can be seriously degraded. This degradation in performance results from the output code testing the test masks against each rule mask with no match until the final else is found. For this reason it is recommended that the else/error action only be performed for very infrequent transactions.

In the event that a programmer is convinced that no else/error action/rule is needed, provisions have been made for a null else/error action entry. To input a null else/error action, the first non-blank and non-tab character

after the 'e' or 'E' in the option section should be the at sign. Note that a null action is not a no-operation. In particular, do not use a null action for the else/error option if a return is really intended. A null action for the else/error action causes the last input rule to become the default action in order to avoid the last test of bit masks for the sake of speed.

```
 1
 2 TABLE SUMMARY FOLLOWS
 3
 4 TABLE NUMBER     1.
 5 number of conditions = 3
 6 number of rules      = 5
 7 number of actions    = 4
 8
 9 CONDITIONS:                          RULES:
10
11 r[c] < 0                          @ y y y y n
12 s[c] < 0                          @ y y n n -
13 t[c] < 0                          @ y n n y -
14
15 ACTIONS:
16
17 printf("T < 0 : ")                @ X     X
18 printf("S < 0 : ")                @ X X
19 printf("R < 0 \n ")               @ X X X X
20 rpos()                            @       X
21 **** NULL ELSE/ERROR ACTION ****
22 Table 1 is complete and non-ambiguous.
```

FIGURE 3. Example of Formatted Output from DELTRANS

3. Action Sequence

When a particular rule is satisfied, the actions designated for that rule are performed in the order listed. If severe programming difficulties should arise from this restriction, the problem may be eased by listing an action in more than one place in the input decision logic table. This will not affect the size of the output execution module.

C. OUTPUT GENERATED

DELTRANS not only generates a coded execution module but also generates a formatted decision logic table for programmer use in documentation and debugging.

Figure 3 contains the formatted output from the example program in Figure 2. This formatted output is written into the standard output. (See section III for a discussion of input and output files.) In Figure 3 the table number is on line 4. Lines 5 through 7 contain a summary of the programmer input for number of rules, actions, and conditions. Lines 9 through 21 display the four quadrants of the decision logic table. Line 22 specifies the else/error action and the last line contains the ambiguity and completeness message. Error messages are also directed to the standard output. See section IV for error messages and suggested

correcting actions. Section III contains procedures for the correction of errors.

The coded execution module is written into the output file. The format of this code, while difficult to understand at first glance, can assist the programmer in debugging syntax errors in input decision logic tables. Figure 4 contains the code generated by DELTRANS from the input program in Figure 2. Line 20 in Figure 2 contains the name and parameters for the subroutine which appear in Figure 4 on line 19. In Figure 4, line 23 contains the declarations for the rule masks (dda), and the test mask(ddb and ddc). Lines 25 through 29 initialize the test mask and rule masks. Lines 30 through 35 provide the code to test the conditions and set the test mask. The remainder of the code searches for a match between a specific rule mask and the test mask, and also contains the actions to be executed if a match is found.

The 'n' or 'N' option placed the subroutine name in the ouput. The 'r' and 'c' options determined the number of tests to be performed and the number of rule masks. The 'a' option merely limits the number of actions to the internal storage requirements in DELTRANS. The final else is used for the final rule. Note that the actions and conditions appear in the output in the same order they have in the input.

```
 1 int r[6]     { 0, 1,-2,-3, 2,-3};
 2 int s[6]     {-4, 9,-7, 0,-4,-1};
 3 int t[6]     {-1, 3,-2,-1, 0, 3};
 4
 5 main () {
 6     int k;
 7     for (k=0; k<6; k++) {
 8         printf("\nThe numbers to test are :\n");
 9         printf("\tR = %d\n\tS = %d\n",r[k],s[k]);
10         printf("\tT = %d\n",t[k]);
11         logtab(k);
12     }
13 }
14
15 rpos () {
16     printf("R is positive.\n");
17 }
18
19 logtab(c)   int c;  {
20     int a;
21     char b;
22
23     int dda[5][2],ddb,ddc;
24     ddb=0;  ddc= 0160000 ;
25     dda[0][0]= 0160000; dda[0][1]= 00;
26     dda[1][0]= 0140000; dda[1][1]= 020000;
27     dda[2][0]= 0100000; dda[2][1]= 060000;
28     dda[3][0]= 0120000; dda[3][1]= 040000;
29     dda[4][0]= 060000;  dda[4][1]= 0160000;
30     if(r[c] < 0 ){
31         ddb =| 0100000; ddc =& 077777;}
32     if(s[c] < 0 ){
33         ddb =| 040000;  ddc =& 0137777;}
34     if(t[c] < 0 ){
35         ddb =| 020000;  ddc =& 0157777;}
36     if((dda[0][0]&ddb)==ddb && (dda[0][1]&ddc)==ddc){
37         printf("T < 0 : ") ;
38         printf("S < 0 : ") ;
39         printf("R < 0 \n ") ;}
40     else if((dda[1][0]&ddb)==ddb && (dda[1][1]&ddc)==ddc){
41         printf("S < 0 : ") ;
42         printf("R < 0 \n ") ;}
43     else if((dda[2][0]&ddb)==ddb && (dda[2][1]&ddc)==ddc){
44         printf("R < 0 \n ") ;}
45     else if((dda[3][0]&ddb)==ddb && (dda[3][1]&ddc)==ddc){
46         printf("T < 0 : ") ;
47         printf("R < 0 \n ") ;}
48     else {
49         rpos()         ;}
50 }
```

FIGURE 4. Example Source Code from DELTRANS

III. EXECUTING YOUR JOB

A. INITIATING DELTRANS

The command line for DELTRANS has the form of "deltrans" followed by the input and output filenames, if desired.

If there are no files specified, DELTRANS will open the standard input for input and create the file "d.tab.c" for output.

If there is only one file name specified, DELTRANS will open that file for input and create the file "d.tab.c" for output.

If there are two filenames specified, DELTRANS will open the first file for input and create a file for ouput using the second filename.

If there are more than two filenames specified in the command line, the first two are used for input and output, and an error message is produced alerting the programmer that the system detected an error in the command line.

If the programmer wishes to keep a record of the formatted table summaries and error messages, he may do so using

the ">" and "|" provided by UNIX for redirecting the standard output to a file or device.

If the code in Figure 2 was in file "fig2", then the command "deltrans fig2 fig4 > fig3" would generate the output code in file "fig4", and redirect the table summary to file "fig3". Figure 4 contains the code that would be generated in file "fig4" and Figure 3 contains the table summary that would be in file "fig3".

Since the output from DELTRANS is C source code and must be compiled prior to execution, file names must be of the form "*.c".

B. ERROR PROCEDURES

DELTRANS is designed to detect a number of errors during execution. A list of error messages is in section IV of this manual along with suggested programmer response to recover from each error. The following discussion is directed at classifying the errors by type.

Ambiguity and completeness checking results in warning messages (program execution continues) when a redundant or incomplete decision logic table is encountered. Inconsistent tables and actionless rules cause error message generation and DELTRANS discontinues processing of the current

table. See chapter V of the parent thesis for a discussion of ambiguity and completeness checking.

File errors are discovered by the system and communicated to DELTRANS for error message generation. All file errors cause immediate termination of preprocessing.

Errors in the option section include both numerical value and syntax. Value errors result when invalid numerals (too large, too small, improperly formed) are encountered. Additionally, missing or duplicate entries for required options cause error message generation and termination of table preprocessing.

Stub errors (else/error, action, and condition) are normally caused by exceeding the limit on stub length or failure to use the delimiter, '@', in the proper place. Table preprocessing is terminated when a stub error is detected.

Action and condition entry errors are caused by either syntax or invalid rule numerals. Both of these are fatal errors in that preprocessing of the current table is discontinued.

Tally errors result when the actual number of input actions or conditions does not agree with the number specified in the option section. Because several arrays, inter-

nal to DELTRANS, have been initialized using the value in the option section, this error causes termination of preprocessing for the current table.

Numerical value errors can occur in many sections of the input. They result from excessively large numerals, excessively small numerals, or improperly formed numerals. The numerals are resized by substituting the largest permitted number for that usage. An error message is generated and DELTRANS attempts to continue execution. However, no output code will be generated, only error checking will be performed.

Few errors cause immediate termination of preprocessing. Normally DELTRANS will attempt to resynchronize the input stream and continue preprocessing in an attempt to generate a formatted table for programmer use in debugging. This attempt to generate assistance for the programmer will include not only the generation of a formatted table, but also will include the generation of some output code whenever possible. However, this is forced output, generated only to assist in debugging. Seldom will executable code result when an error has been detected.

C. CHANGING THE INPUT DATA

Since DELTRANS was constructed to preprocess a file or input with multiple decision logic tables imbedded in that input, there is no method available to the programmer to alter the preprocessor actions other than alter the input data and preprocess the data again. For this reason the programmer is strongly encouraged to prepare the input as a file rather than enter it from the terminal.

D. COMPILATION AND EXECUTION

The output file generated by DELTRANS contains both the C code from the input program as well as the C code generated by the preprocessor. In order to execute the program it is necessary to compile the output program. See the applicable UNIX reference manual for detailed instructions for compiling and executing C programs.

IV. ERROR MESSAGES

A user may receive one or more error messages at his terminal during preprocessing. They may cause immediate processing termination or may only be a warning to point out possible faulty logic. In either case, this section may be used as a guide for error identification and correction.

All possible DELTRANS error messages are presented here as they will appear at the terminal, along with an explanation and required user response. In the event any other messages appear at the terminal, the UNIX Progammer's Manual [26] and the C Reference Manual [23] should be consulted.

A. WARNING MESSAGES

A list of processor generated warning messages follow. Those not considered self-explanatory are given an error number for reference.

WARNING ***** duplicate entries for number of actions.

WARNING ***** duplicate entries for number of conditions.

WARNING ***** duplicate entries for number of rules.

WARNING ***** option value less than or equal to zero in line X.

WARNING ***** duplicate rules specified for action Y line X.

WARNING *A02* rules X and Y in table Z are redundant.

WARNING *A03* table Z is incomplete.

B. AMBIGUITY AND COMPLETENESS ERRORS

A01: ERROR *A01* rules X and Y make table Z inconsistent.

EXPLANATION:

In table number Z, rules X and Y can be specified by the same set of conditions. However, they require different actions and thus make the table logic faulty. Further processing on table Z was terminated.

RESPONSE:

1. Examine carefully rules X and Y. Note that all "don't cares" ('-') can be expanded to both a "y" and a "n" entry. Also a "*" is equivalent to a "n" and a "$" is equivalent to a "y".
2. Alter rules X and Y to remove the logic error.

A02: WARNING *A02* rules X and Y make table Z redundant.

EXPLANATION:

In table number Z, rules X and Y can be satisfied by the same set of conditions. Furthermore, they specify the same action and thus either one rule can be removed or the two rules combined. Table processing was not terminated due to this warning.

RESPONSE:

Remove the redundancy by either removing one rule or combining the two.

A03: WARNING *A03* table Z is incomplete.

EXPLANATION:

This warning is generated to alert the user that the logic in table number Z does not consider all possible combinations of conditions. In addition, the else/error action is a null action. Table processing was not terminated due to this warning.

RESPONSE:

1. Ensure all missing rules would only be satisfied by impossible condition outcomes.
2. If it can not be shown that all missing rules are impossible, enter an appropriate error action.
3. If all missing rules are impossible, ignore the warning. The table output follows prescribed logic.

A04: ERROR *A04* no action specified for rule X.

EXPLANATION:

Internal ambiguity checks reveal that rule X has no associated actions.

RESPONSE:

1. Recheck format and logic.
2. To implement a true no-action rule, input a null action and specify that action for rule X.

C. FILE ERRORS

F01: ERROR *F01* unable to open 'd.tab.c' for output.

EXPLANATION:
UNIX could not open the default output file "d.tab.c" for output. File errors cause termination of preprocessing.

RESPONSE:
1. Ensure prooer format of translator call line.
2. If there are no other program errors, contact a consultant.

F02: ERROR *F02* unable to open FILE for input.

EXPLANATION:
UNIX could not open the given file for input. File errors cause termination of processing.

RESPONSE:
1. Ensure proper format of translator call line.
2. Ensure given file exists.
3. If there are no other program errors, contact a consultant.

F03: ERROR *F03* unable to open FILE for output.

EXPLANATION:
UNIX could not open the given file for output. File errors cause termination of processing.

RESPONSE:
1. Ensure proper format of translator call line.
2. If there are are no other program errors contact a consultant.

D. RULE NUMBER IN ACTION OR CONDITION ENTRY

L01: ERROR *L01* line X.

EXPLANATION:
The action entry on line X specifies that that action is to be performed for a rule number that is greater than the maximum number of rules declared in the option section. Processing will be terminated for this table if the error count exceeds the maximum allowable.

RESPONSE:
1. Check the indicated line to ensure the format is correct.
2. Check the largest number in the line against the number of rules specified in the program option section.

L02: ERROR *L02* line X.

EXPLANATION:
A condition entry specifies a condition is to be tested for an invalid rule number. Processing will be terminated for this table if the error count exceeds the maximum allowable.

RESPONSE:
1. Check that all rule numbers in the specified entry line are no larger than the maximum specified in the option section.

2. Examine line X for format errors.

L03: ERROR *L03* line X.

EXPLANATION:
A condition entry contains an invalid list of rule. Processing will be terminated for this table if the error count exceeds the maximum allowable.

RESPONSE:
1. Check for proper format of line X.
2. Ensure all rule numbers are greater than zero.
3. Ensure all rule numbers are no larger than the maximum rule number specified in the option section.
4. Ensure the second number in a list is greater than the first.

L04: ERROR *L04* line X.

EXPLANATION:
An action or condition entry specifies a rule number that is not in the specified range.

RESPONSE:
Check that all rule numbers in the specified entry line are no larger than the maximum specified by the program.

E. OPTION SECTION ERRORS

N01: ERROR *N01* initialization error.

EXPLANATION:
The number of rules, actions, and conditions must all be initialized in the option section. In addition they must all be greater than zero.

RESPONSE:
Ensure that all rules, actions, and conditions are initialized and in the proper format. The number must follow the option on the same line as the option.

N02: ERROR *N02* missing subroutine name.
ERROR *N02* subroutine name missing from options.
ERROR *N02* subroutine name exceeds 8 characters near line X.

EXPLANATION:
The option section for each decision logic table must contain the option "n" which specifies the name of the subroutine into which the decision logic table will be converted. Processing of the current table is halted when unable to find the subroutine name.

RESPONSE:
1. Check for proper format of the "n" option.

2. Ensure the name is no longer than 8 characters and is separated from the parameter list by a space.
3. Ensure the name is on the same line as the "n".

N03: ERROR *N03* unrecognizable option 'Q' line X.

EXPLANATION:
An unrecognizable option has been detected on line X. 'Q' is the character DELTRANS found when it was expecting an option character.

RESPONSE:
1. Ensure the proper format is used. Notice that no punctuation, other than spaces, is valid in the option section.
2. See also error N04.

N04: ERROR *N04* invalid numeral 'Q' near 'P' line X.

EXPLANATION:
An invalid numeral has been detected on line X following P.

RESPONSE:
1. Ensure that the proper option format is used, including the circumflex.
2. The number must follow the option on the same line as the option.
3. When the logic table translator drops synchronization in the option section, error N03 and N04 are repeated while resyncronization is attempted. This is an indication of invalid option letters, format, or punctuation.

N05: ERROR *N05* invalid number of rules line X set to Y.
ERROR *N05* invalid number of actions line X set to Y.
ERROR *N05* invalid number of conditions line X set to Y.

EXPLANATION:
The option section of the input program requested that either the number of actions, conditions, or rules exceed the maximum permitted by the translator.

RESPONSE:
1. A series of small, linked tables are recommended.
2. Check line X to ensure the option size is as needed.
3. A consultant can enlarge the processor for special applications.

N06: ERROR *N06* declarations found prior to subroutine name.

EXPLANATION:
The option section must contain the "n" toggle prior to the "d" toggle to perform proper coding. No further processing on the current table is attempted.

RESPONSE:
Edit the option section of the table and place the "n" toggle before the "d" toggle.

N07: ERROR *N07* invalid parameter list.

EXPLANATION:
The format of the parameter list is in error. The translator has recognized and accepted the name of the subroutine but is unable to find a valid list of parameters. Output has been forced into the specified output file as a possible aid in debugging.

RESPONSE:
Locate the 'n' toggle in the option section of the table and correct the format error. Note that a "{" must be included.

N08: ERROR *N08* else/error syntax line X.
ERROR *N08* missing else/error action line X.

EXPLANATION:
ELSE/ERROR actions are required for every decision logic table to be processed by DELTRANS. If no action is desired a "e@" will set a null action. Proper format requires that the action be on the same line as the toggle "e".

RESPONSE:
Correct the format error on line X.

F. ACTION AND CONDITION STUB ERRORS

S01: ERROR *S01* invalid else/error stub line X.
ERROR *S01* invalid action stub line X.
ERROR *S01* invalid condition stub line X.

EXPLANATION:
An improperly formed stub has been detected. Table processing is terminated when this error occurs.

RESPONSE:
Ensure proper format of the stub on line X, in particular that the "@" is in position and that the stub contains no more than 31 characters.

S02: ERROR *S02* invalid condition stub line X.

EXPLANATION:
Empty condition stubs are invalid. The output code will fail to compile since a test of a null condition would be required.

RESPONSE:
Check the format on line X.

G. ACTION AND/OR CONDITION TALLY ERRORS

T01: ERROR *T01* number of actions not as specified in option section.

EXPLANATION:
Internal checks indicate the actual number of actions input does not equal the number of actions specified in the option section. Table processing has been terminated after forcing table summary output.

RESPONSE:
Check the number of actions and the format of the option statement. The output table will aid in locating the error.

T02: ERROR *T02* number of conditions not as specified in option section.

EXPLANATION:
Internal checks indicate the actual number of conditions input does not equal the number of conditions specified in the option section. Table processing has been terminated.

RESPONSE:
Ensure that the option number and the format are correct. See the output table for summary debugging assistance.

H. NUMERICAL VALUE ERRORS

V01: ERROR *V01* excessive value line X set to MAX.

EXPLANATION:
An integer larger than any valid value in the translator has been detected. Further processing will be attempted.

RESPONSE:
1. Check integer size on line X.
2. Ensure proper format up to and including line X.
3. See error N05.

V02: ERROR *V02* invalid numeral 'D' line X.

EXPLANATION:
An improperly formed string of numerals has been encountered.

RESPONSE:
Ensure that all strings of numerals are in proper format and that no characters other than integers appear in the string.

V03: ERROR *V03* line X.

EXPLANATION:
A non-existent rule number appears in the action entry on line X.

RESPONSE:
Check all rule numbers in action entries to ensure that they are greater than zero and are of the proper format.

V04: ERROR *V04* line.

EXPLANATION:
A non-existent rule number appears in the condition entry on line X.

RESPONSE:
Ensure all entries refer to positive non-zero rules. Check the format of the condition entry.

I. SYNTAX ERRORS

X01: ERROR *X01* statement syntax line X.

EXPLANATION:
A syntax error was discovered by DELTRANS in line X.

RESPONSE:
Review the format up to and including the line indicated.

X02: ERROR *X02* statement syntax line X.

EXPLANATION:
A syntax error was discovered by DELTRANS in line X. At the time of the error, DELTRANS was attempting to decode a condition entry list.

RESPONSE:
Review the format up to and including the line indicated.

X03: ERROR *X03* statement syntax line x.

EXPLANATION:
A syntax error was discovered by DELTRANS in line X. At the time the error was detected DELTRANS was attempting to decode an action entry list.

RESPONSE:

Review the action list format up to and including the line indicated.

V. GLOSSARY OF TERMS

Action: Something to be done predicated on the responses to the conditions in the table. May be computations, goto statement, assignment, etc.

Action Entry: The lower right quadrant of the table. The only entry permitted in this section for tables in limited entry format is an "X". When an "X" is placed opposite an action, that action is to be taken.

Action Line: One action from the action stub and its associated list of rules from the action entry.

Action Statement(s): The contents of the action stub.

Action Stub: The lower left quadrant of the body of the table. Listed here are the actions to be taken, which depend on the conditions in the condition stub above.

Complex Rule: A decision table rule which contains at least one "don't care" entry.

Condition: A test or a decision to be made as part of the logic or processing of a problem. It should be stated in a form that may be answered "yes" or "no".

Condition Entry: The upper right quadrant of the body of the decision table. This section contains responses to questions asked in the condition stub.

Condition Line: One condition from the condition stub and its associated list of truth values for rules from the condition entry.

Condition Statement(s): Contents of the condition stub.

Condition Stub: The upper left quadrant of the body of the decision table. All the conditions or tests to be made will appear in this section.

Contradiction: A decision logic error in which two or more rules have the same combination of

conditions but with different actions specified. A synonym for inconsistent.

ELSE / ERROR Rule: The rule that acts as a catch-all for all rules not specifically covered in the table. Its presence completes the table.

Extended Entry: A type of decision table in which actual condition values are specified in the condition entry part of the table.

GOTO: Used for linking tables, it is an action statement which references another table.

Impossible Rule: A rule that due to dependency of conditions is physically impossible, e.g. c < 10 and c > 20.

Inconsistency: See contradiction.

Limited entry: A type of decision table in which all conditions are stated as questions which have a yes or no answer.

Mixed Entry: A type of decision table in which both limited entries and extended entries appear.

Recursion: A synonym for looping. A conditional branch is made, depending on a condition value.

Redundancy: A decision logic error in which two or more rules have the same combination of conditions with the same actions specified.

Rule: A single column of the decision table that shows the combination of responses to the conditions and the resulting or appropriate actions.

Rule Mask: A method of program coding using decision tables, where a table matrix is held in storage and data matched against it.

Simple Rule: A rule which contains no "don't care" entries.

Table Linkage: A conversion by which two or more tables are related by means of action transfers (such as GOTO) from one to another.

APPENDIX B - DELTRANS SOURCE CODE

```
%token COND ACT TLIST ALIST DIGIT NUM
%{
#
#define ACTLEN   32   // max chars in act stub
#define BIGINT   64   // max integer in processor
#define BLANK    ' '
#define CDNLEN   32   // max chars in cdn stub
#define CRLF     '\n'
#define DELIM    '@'
#define EOF      -1   // end of file from getc()
#define ERRLEN   32   // max chars in else/error action
#define GNC      getc(inp)  // used for program clarity
#define MAXRULE  64   // maximum number of rules
#define MAXCDN   16   // maximum number of conditions
#define MAXACT   32   // maximum number of actions
#define MAXERR   5    // maximum number of non-fatal errors
#define NULL     '\0'
#define TAB      '\t'
#define TOKN     '←'
#define TRUE     1
#define FALSE    0

int enum     0;      // number of errors
int inconsis FALSE;  // table inconsistency flag
int line     1;      // line number
int nexta    0;      // index into action structure
int nextc    0;      // index into condition structure
int nodec    TRUE;   // negative declaration flag
int noelse   TRUE;   // no error/else rule flag
int noname   TRUE;   // no subroutine name found
int numact   0;      // number of actions
int numcdn   0;      // number of conditions
int numrule  0;      // number of rules
int parsact  COND ;  // parse action flag
int pbak     -2;     // aux next character buffer
int peek     -2;     // next character buffer
int sumact   0;      // check sum on number of actions
int sumcdn   0;      // check sum on number of conditions
int tabno    0;      // current table number
```

```
int rule[MAXRULE];      //  pointers to required actions
                        //       for each rule

int rmask[MAXRULE][2];      //  rule mask matrix

char eeact[ERRLEN] ;    // error/else action


struct {                // condition stubs and condition
   char    cltr[CDNLEN];   // entries
   char    centry[MAXRULE];
}cstub[MAXCDN], *cptr;


struct {                // action stubs and action entries
   char    altr[ACTLEN];
   char    aentry[MAXRULE];
}astub[MAXACT], *aptr;


struct {                // rule action lists
   int     actptr;
   int     actltrs;
}free[BIGINT], *fptr;


struct iobufr {             // both the input and output
   int iobfd ;          // buffers
   int ioleft;
   char *ionxtp ;
   char iobuff[512] ;
   } inbuf, outbuf, *inp, *outp;

%}
%%
detab :
   conhalf acthalf ;

conhalf :
   conline |
   conhalf conline ;

conline :
   conpart cdnlist ;

conpart :
   COND = {getcdn();};

cdnlist :
   clist ';' = {cdntest();};
```

```
lead :
   logltr '(' ;

cdig :
   lead DIGIT = { fill($2); $$ = $2; } |
   coma DIGIT = { fill($2); $$ = $2; } ;

cdash :
   cdig '-' DIGIT = { fillup($1,$3) ;  } ;

coma :
   cdig ',' |
   cdash ',' ;

clist :
   |
   clist cdig ')' |
   clist cdash ')' ;

logltr :
   'y' = { peek = 'y' ; }|
   'Y' = { peek = 'y' ; }|
   'n' = { peek = 'n' ; }|
   'N' = { peek = 'n' ; }|
   '*' = { peek = '*' ; }|
   '-' = { peek = '-' ; }|
   '$' = { peek = '$' ; };

acthalf :
   actline |
   acthalf actline;

actline :
   actpart actlist ;

actpart :
   ACT = {getact();};

actlist :
   nlist ';' = { actest(); } ;

nlist :
   NUM = {addrule($1,&astub[nexta-1]);
          parsact=ALIST;}    |
   pal NUM = {addrule($2,&astub[nexta-1]);
              parsact = ALIST ; } ;

pal :
   nlist ',' ;

%%
```

```
actest()   {        /* called at the end of each action
                       entry list to determine what step
                       the preprocessor should take next */
   if ((peek=eatall()) != TOKN && sumact < numact)
       parsact = ACT;
   else if (peek==TOKN && sumact==numact)
       parsact = NULL;
   else
       sumerr("*T01*","actions");
}

addrule (x,y) int x, y; {       /* maintans rule pointer
                                   list ("rule"), rule action
                                   list ("free"), and action
                                   entries in astub   */
   int ptr;
   char *c ;
   if (x > numrule) badlogic("L01");
   else if (x <= 0) badlogic("V03");
   else if (*(c= &aptr->aentry[x-1])!='X') {
       *c='X';
       fptr -> actltrs = y;
       if (rule[x])    {
           ptr=fptr;
           fptr=rule[x];
           while (fptr -> actptr)
               fptr=fptr -> actptr;
           fptr -> actptr = ptr;
           fptr=ptr;
       }
       else
           rule[x]=fptr;
       ++fptr;
   }
   else    {
       printf("WARNING  *****  duplicate rules ");
       printf("specified for action ");
       printf("%d line %d.\n",x,line);
}  }
```

```
ambigck()  {                    /* driver for ambiguity
                                and completeness testing */
    int noambig, c, k;
    noambig = TRUE;
    for (c=0; c < (numrule -1); c++)     {
        if (goodrule(c))
            for (k= c+1; k < numrule; k++)  {
                if (samerule(c,k))  {
                    ambigtyp(c,k);
                    noambig = FALSE;
                }
            }
        else
            noambig = FALSE;
    }
    if (goodrule(c) && noambig)
        complete();
    return(inconsis);
}

ambigtyp(c,k)  int c, k;    {   // determines type of
    int n;                      // ambiguity that exists
    for (n=0; n < numact ; n++) {
        if (astub[n].aentry[c] != astub[n].aentry[k]) {
            printf(" ERROR   *A01*  Rules %d ",++c);
            printf("and %d make table %d ",++k,tabno);
            printf("inconsistent.\n");
            inconsis = TRUE;
            return;
        }
    }
    printf("WARNING  *A02*  Rules %d and %d ",++c,++k);
    printf("in table %d are redundant.\n",tabno);
}

badlogic(c)    char    *c; {        // logic error routine
    printf(" ERROR   *%s* line %d.\n",c,line);
    if (++enum > MAXERR) bomb();
}

bomb () {                       // excessive error exit
    printf("Processing of table %d halted ",tabno);
    printf("due to error count.\n");
    killex();
}
```

```
cdntest () {                    // called at the end of each
                                // condition entry list to
                                // determine what step the
                                // preprocessor should take
   if ((peek=eatall())!='↑' && sumcdn < numcdn)
       parsact = COND;
   else if (peek=='↑' && sumcdn==numcdn)   {
       parsact = ACT;
       peek = eatall();
   }
   else
       sumerr("*T02*","conditions");
}

complete() {            // called to determine if an
   // unambiguous table is complete.  Completeness
   // testing of an ambiguous table is ........
   int c, k;                    // ........ ambiguous
   k=0;
   for (c=0; c < numrule; c++)
       k =+ totrule(c);
   if (k == (1 << numcdn)) {
       printf("Table %d is complete and ",tabno);
       printf("non-ambiguous.\n");
       if (eeact[0]!=NULL) {
           printf("The else/error rule for this table ");
           printf("is suoerfluous.\n");
       }
   }
   else if (eeact[0]==NULL)    {
       printf("WARNING  *A03*  table %d is ",tabno);
       printf("incomplete.\n");
}  }
```

```
copycode(ltr)  char ltr ;  {    // copy code to output file
   int k, z;                    // up to but excluding ltr
   while((k=GNC) > EOF)     {
       if (k == ltr)
           return(TRUE);                  // normal exit
       switch (k)  {
           case '"' : case '\'' :  // direct copy of quotes
               putc(k,outp);
               while((z=GNC) > EOF && z != k)  {
                   if(z==CRLF) line++;
                   putc(z,outp);
               }
               if(z == EOF) thatsit();
               putc(z,outp);
               break;
           case '/' :                 // comments are removed
               k=GNC;
               switch (k)  {
                   case EOF :              // EOF
                       thatsit();
                   case '/' :              // strip rest
                       tossline(k);        //    of line
                       putc(CRLF,outp);
                       break;
                   case '*' :              // strip to end
                       cutcom();           //   of comment
                       break;
                   case CRLF :
                       line++;
                   default :               // normal case
                       putc('/',outp);
                       putc(k,outp);
               }
               break;
           case CRLF :                     // count lines
               line++;
           default :                       // normal case
               putc(k,outp);
   }   }
   return(FALSE);
       // abnormal exit ( fail on EOF )
}
```

```
cutcom ()  {            // strip off comments up to '*/'
   int c   ;
   while ((c=GNC) > EOF)   {
       if (c==CRLF) line++;          // count lines
       else while (c=='*') {         // end of comment ?
           if ((c=GNC)==CRLF) line++;
           else    {
               if (c=='/') return;
               if (c== EOF) thatsit();
   }   }   }
   thatsit();                    // EOF
}

decodacr(p,max,count)                // action, condition,
   char *p;                          // and rule option
   int max, *count;    {             // number decoder
   int num;
   if (*count != 0) {
       printf("WARNING  *****  duplicate entries ");
       printf("for number of %s.\n",p);
   }
   num = gobble();                   // number follows option
   if(num < '0' || num > '9')  {     // if not error
       printf(" ERROR   *N04*  invalid ");
       printf("numeral '%c' near %c ",num,*p);
       printf("line %d.\n",line);
       if(++enum > MAXERR) bomb();
       peek = num;
       return;
   }
   *count = getval(num);
   if(*count > max)    {
       printf(" ERROR   *N05*  invalid number of %s ",p);
       printf("line %d set to %d.\n",line,max);
       *count = max;
       if(++enum > MAXERR) bomb();
   }
   if (*count <= 0)    {
       printf("WARNING  *****  option value less than ");
       printf("or equal to zero in line %d.\n",line);
}  }

eatall () {             // eat up blanks, tabs, and newlines
   int c ;
   while ((c=gobble()) == CRLF )
       ;
   return(c);          //  return the first character
}
```

```
eatc() {                              // buffered eatall
   int c ;                            // used in yylex()
   if(pbak<0 || pbak==CRLF || pbak==TAB || pbak==BLANK)
       c=eatall();
   else
       c=pbak;
   pbak = -2;
   return(c);
}

elserror() {                          // decode else/error action
                                      // and place in eeact[]
   if((eeact[0]=gobble()) == DELIM)
       eeact[0] = NULL;
   else if (eeact[0] == CRLF)  {
       printf(" ERROR   *N08*  else/error syntax ");
       printf("line %d.\n",line);
       if (++enum > MAXERR) bomb();
       return;
   }
   else
       strcopy(ERRLEN -1,&eeact[1],"else/error");
   noelse = FALSE;
}

faterr(h,ptr) int h; char *ptr; {  // fatal error routine
   switch (h) {
       case 1 :
           printf(" ERROR   *F01*  unable to open ");
           printf("'d.tab.c' for output.\n");
           break;
       case 2 :
           printf(" ERROR   *F02*  unable to open ");
           printf("'%s' for input.\n",ptr);
           break;
       case 3 :
           printf(" ERROR   *F03*  unable to open ");
           printf("'%s' for output.\n",ptr);
   }
   printf("Execution terminated due to file error.\n");
   exit();
}
```

```
fcheck()   {                        // check for proper options
   char *u;
   if (numrule==0 || numcdn==0 || numact==0)   {
       u = &("\tThe number of ");
       printf(" ERROR   *N01*  initialization error.\n");
       printf("%srules is      %d.\n",u,numrule);
       printf("%sconditions is %d.\n",u,numcdn);
       printf("%sactions is    %d.\n",u,numact);
       enum = MAXERR +1;
   }
   if (noname) {
       printf(" ERROR   *N02*  missing subroutine name.");
       printf("\n");
       enum = MAXERR +1;
   }
   if (noelse) {
       printf(" ERROR   *N08*  missing else/error ");
       printf("action line %d.\n",line);
       if (++enum > MAXERR) bomb();
   }
   if (enum > MAXERR) bomb();
   peek = eatall();
}

fill (c) int c ; {          // insert logic letter into
   if (c > numrule) badlogic("L02");   //   centry
   else if (c <= 0) badlogic("V04");
   else    cptr -> centry[c - 1] = peek;
}

fillup (a,b) int a,b ; {
       // insert logic letter into centry list
   int i;
   if (a > 0 && a <= b && b <= numrule) {
       for(i = a; i <= b; i++)
           cptr -> centry[i - 1] = peek;
   }
   else    {
       printf("Invalid condition entry: '%d-%d'\n", a, b);
       badlogic("L03");
}  }
```

```
findacr()  {                    // driver for option decoding
   int c ;
   while ((c=gobc()) > EOF) {
       switch (c)  {
           case CRLF :                  // end of line
               break;
           case '/' :                   // rest of line is
               tossline(c);             //       a comment
               break;
           case 'n' : case 'N' :// subroutine name follows
               namsub();
               break;
           case 'd' : case 'D' :// only declarations remain
               if (noname)      {
                   printf(" ERROR   *N06*  declarations ");
                   printf("found prior to subroutine na");
                   printf("me.\nExecution terminated.\n");
                   killex();
               }
               peek = '↑';
               if (copycode(peek)) {
                   nodec = FALSE;
                   break;
               }
               thatsit();               // EOF encountered
           case 'r' : case 'R' :    // options
               decodacr("rules",MAXRULE,&numrule);
               break;
           case 'c' : case 'C' :
               decodacr("conditions",MAXCDN,&numcdn);
               break;
           case 'a' : case 'A' :
               decodacr("actions",MAXACT,&numact);
               break;
           case '↑' :               // end of section
               fcheck();            // check options all defined
               return;
           case 'e' : case 'E' :    // else/error action
               elserror();
               break;
           default :
               printf(" ERROR   *N03*  unrecognizable ");
               printf("option '%c' line %d.\n",c,line);
               if(++enum > MAXERR) bomb();
   }   }
   thatsit();                       // unexpected end of file
}
```

```
fsuberr(k) int k   ;    {   // error in subroutine name
   printf(" ERROR   *N02*  ");
   switch (k)  {
       case 1  :
           printf("subroutine name missing from options");
           printf(".\n\nExecution terminated.\n");
           killex();
       case 2  :
           printf("subroutine name exceeds 8 characters ");
           printf("near line ");
           printf("%d.\n",line);
           if(++enum > MAXERR) bomb();
}  }

gencode()  {            // driver for table output
   genmasks();          // code for declarations
   geninit() ;          // code to initialize masks
   gsetmask();          // code to test conditions
                        // and set mask
   genstats();          // output if-else statments
   geneot()  ;          // output end of table
}

gendub()   {                    // output CRLF TAB TAB
   putc(CRLF,outp);
   putc(TAB,outp);
   putc(TAB,outp);
}

genelse()  {                    // output else action
   gensig();
   outcode("else {");
   if (eeact[0])   {
       gendub();
       outcode(&eeact[0]);
       putc(';',outp);
   }
   else    {
       fptr = rule[numrule];
       while (fptr)    {
           gendub();
           outcode(fptr->actltrs);
           putc(';',outp);
           fptr = fptr->actptr;
   }   }
   putc('}',outp);
}
```

```
geneot()   {                        // generate final "{" for
   putc(CRLF,outp);                 // output subroutine
   putc('}',outp);
   putc(CRLF,outp);
}

geninit()  {              // output code to initialize masks
   int c   ;
   gensig();
   outcode("ddb=0;");
   putc(TAB,outp);
   outcode("ddc= ");
   outoct(0177777 << (16 - numcdn));
   outcode(" ;");
   for (c=0; c < numrule; c++) {
       gensig();
       outcode("dda[");
       outdec(c);
       outcode("][0]= ");
       outoct(rmask[c][0]);
       putc(';',outp);
       putc(TAB,outp);
       outcode("dda[");
       outdec(c);
       outcode("][1]= ");
       outoct(rmask[c][1]);
       putc(';',outp);
}  }

genmasks() {                        // output mask declarations
   gensig();
   outcode("int dda[");
   outdec(numrule);
   outcode("][2],ddb,ddc;");
}

gensig()   {                        // output CRLF TAB
   putc(CRLF,outp);
   putc(TAB,outp);
}
```

```
genstats() {                        // output if-else statments
    int k, nr   ;
    if (eeact[0])
        nr = numrule;
    else
        nr = numrule -1;
    gensig();
    outcode("if((dda[0][0]&ddb)==ddb && ");
    outcode("(dda[0][1]&ddc)==ddc){");
    fptr=rule[1];
    while (fptr)    {
        gendub();
        outcode(fptr->actltrs);
        fptr=fptr->actptr;
        putc(';',outp);
    }
    putc('}',outp);
    for (k=1; k < nr ; k++) {
        fptr=rule[k+1];
        gensig();
        outcode("else if((dda[");
        outdec(k);
        outcode("][0]&ddb)==ddb && ");
        outcode("(dda[");
        outdec(k);
        outcode("][1]&ddc)==ddc){");
        while (fptr)    {
            gendub();
            outcode(fptr->actltrs);
            putc(';',outp);
            fptr=fptr->actptr;
        }
        putc('}',outp);
    }
    genelse();
}
```

```
getact () {              //  fetch and store the next action
    int j;
    char *p;
    p = &astub[nexta].altr[0];
    aptr = &astub[nexta++];
    *p = peek;
    peek = -2;
    if (*p++ != DELIM)
        strcopy(ACTLEN -1,p,"action");
    else
        *(--p) = NULL;
    sumact++;
    parsact = ALIST;
    for (j=0; j < numrule; j++)
        aptr->aentry[j] = BLANK ;
}

getcdn () {              //  fetch and store next condition
    int j;
    char *p;
    p = &cstub[nextc].cltr[0];
    cptr = &cstub[nextc++];
    *p = peek;
    peek = -2;
    if (*p == DELIM)     {
        printf(" ERROR   *S02*  ");
        printf("Invalid condition stub line %d.\n",line);
        killex();
    }
    strcopy(CDNLEN -1,++p,"condition");
    sumcdn++;
    parsact = TLIST;
    for (j=0; j < numrule; j++)
        cptr->centry[j] = '-';
}
```

```
getval (c) int c ; {    // decipher a string of numerals
   int num ;
   num = c - '0' ;
   while ((c=GNC) >= '0' && c <= '9')  {
       num = num * 10 + c - '0' ;
       if (num > BIGINT) {
           printf(" ERROR   *V01*  excessive value ");
           printf("line %d set to %d.\n",line, BIGINT);
           if (++enum > MAXERR) bomb();
           while((c=GNC)!=BLANK && c!=TAB && c!=CRLF)
               if ( c == EOF ) thatsit();
                       // find end of number
           if(c==CRLF) line++;
           return(BIGINT);
   }   }
   if ( c == EOF ) thatsit();          // End-of-file
   if (c==BLANK || c==TAB) return(num);
   if (c==CRLF) {
       line++;
       return(num);
   }
   printf(" ERROR   *V02*  inva");          // error exit
   printf("lid numeral : '%d%c' line %d.\n",num,c,line);
   if (++enum > MAXERR) bomb();
   peek = c;
   return(num);
}

gobble () {             // find first character other than
   int c ;      //      a blank or tab
   while ((c=GNC)==BLANK || c==TAB)
       ;
   if (c==CRLF) line++;
   else
       if (c == EOF) thatsit();
   return(c);
}

gobc() {                        // buffered gobble
   int c;                       // used in findacr()
   if(peek < 0 || peek==TAB || peek==BLANK)
       c=gobble();
   else
       c=peek;
   peek = -2;
   return(c);
}
```

```
goodrule(c)     int c ; {   // each rule must have at least
    int k ;                 // one action
    for (k=0; k < numact; k++)  {
        if (astub[k].aentry[c]=='X')
            return(TRUE);
    }
    inconsis = TRUE;
    printf(" ERROR   *A04*  no actions specified ");
    printf("for rule %d.\n",++c);
    return(FALSE);
}

gsetmask() {                // output code to test conditions
    int c, k;               // and set test mask
    for (c=0; c < numcdn; c++)  {
        gensig();
        outcode("if(");
        outcode(cstub[c].cltr);
        outcode("){");
        gendub();
        outcode("ddb =| ");
        outoct(k=(01 << (15-c)));
        putc(';',outp);
        putc(TAB,outp);
        outcode("ddc =& ");
        outoct(~k);
        putc(';',outp);
        putc('}',outp);
}   }

initvar() {                             // initialize variables
    fptr = free;
    inp  = &inbuf.iobfd ;
    outp = &outbuf.iobfd ;
}

killex()   {                        // kill execution
    int c   ;
    fflush(outp);
    while ((c=eatall()) != TOKN)    ;   // attempt resync
    run();
}

main(argc,argv) int argc ; char ** argv ;  {
    namfile(argc,argv);
    if (copycode(TOKN)) {
        proctab();
        run();
    }
    fflush(outp);
}
```

```
namend()   {                    // check for end-of-name
    int c   ;
    if ((c=GNC) == EOF) thatsit();
    if (c==BLANK || c==TAB)     return(-1);
    if (c==CRLF)    {
        linett;
        return(-1);
    }
    return(c);
}

namfile (argc,argv)             // fetch names for files
    int argc ;              // from command line
    char *argv[] ; {
    int k ;
    initvar();
    if(argc < 2) {
        if((k = fcreat("d.tab.c",outp)) == -1)
            faterr(1,argv[0]);
    }
    else if(argc == 2) {
        if((k = fcreat("d.tab.c",outp)) == -1)
            faterr(1,argv[1]);
        if((k = fopen(argv[1],inp)) == -1)
            faterr(2,argv[1]);
    }
    else {
        if((k = fopen(argv[1],inp)) == -1)
            faterr(2,argv[1]);
        if((k = fcreat(argv[2],outp)) == -1)
            faterr(3,argv[2]);
        if (argc > 3) {
            printf("Trash in command line :: %s\n",argv[3]);
}   }   }
```

```
namsub()   {            //  copy subroutine name to output
   int c, k    ;
   if ((c=gobble())==CRLF) fsuberr(1);     // missing name
   putc(c,outp);
   noname = FALSE;
   for (k=1; k < 9 ; k++)  {
       if ((c=namend()) < 0)   {
               // negative return indicates end
           parmlist();                  // copy parameters
           return;                      // normal return
       }
       putc(c,outp);
   }
   fsuberr(2);                      // name too long
   while((c=namend()) >= 0) ;
   parmlist();
}

outate(c)  int c   ;   {        // output octal digits
   int k   ;
   if (k = (c>>3))
       outate(k);
   putc((c%8 + '0'),outp);
}

outcode(p) char *p ;   {        // output a string
   while(*p)
       putc(*p++,outp);
}

outdec(c)  int c   ;   {        // output a decimal
   int k   ;
   if (k = c/10)
       outdec(k);
   putc((c%10 + '0'),outp);
}

outoct(c)  int c   ;   {        // output an octal number
   int k, t    ;
   putc('0',outp);
   if(c>=0) outate(c);
   else    {
       putc('1',outp);
       c =& (t=077777);
       for (k=12; k>=0; k=k-3) {
           putc(((c>>k) + '0'),outp);
           c =& (t=(t>>3));
}  }   }
```

```
pact (x) int x; {           // print action stub and
    int j;                  //   entry list
    j = 0;
    printf("\n%-32.32s@", astub[x].altr);
    while(j <= (numrule - 1))
        printf(" %c", astub[x].aentry[j++]);
}

parmlist() {            // copy parameter list to output
    int c   ;
    while ((c=GNC) > EOF && c != '{')   {
        putc(c,outp);
        if (c==CRLF)    line++;
    }
    if (c=='{') {
        putc('{',outp);
        putc(CRLF,outp);
        return;
    }
    printf(" ERROR   *N07*  invalid parameter list.\n");
    thatsit();
}

pcond (x) int x; {                  // print condition stub
    int j;                          // and entry list
    j = 0;
    printf("\n%-32.32s@", cstub[x].cltr);
    while(j <= (numrule - 1))
        printf(" %c", cstub[x].centry[j++]);
}

proctab()  {                    // process table
    tabno++;
    findacr();
    if (yyparse())
        return;
    yyaccpt();
}
```

```
ptable () {                          // driver to print table
   int i;
   printf("\nTABLE SUMMARY FOLLOWS\n");
   printf("\nTABLE NUMBER      %d.\n",tabno);
   printf("number of conditions = %d\n", numcdn);
   printf("number of rules      = %d\n", numrule);
   printf("number of actions    = %d\n", numact);
   printf("\n%s%29s\n", "CONDITIONS:",  "RULES:");
   for(i = 0; i <= numcdn - 1; i++) pcond(i);
   printf("\n\nACTIONS:\n");
   for(i = 0; i <= numact - 1; i++) pact(i);
   printf("\n");
   if (noelse)
       printf("**** MISSING ELSE/ERROR ACTION ****");
   else if (eeact[0])
       printf("ELSE/ERROR ACTION : %s",&eeact[0]);
   else
       printf("**** NULL ELSE/ERROR ACTION ****");
   printf("\n");
}

reinit()   {                         // reinitialize variables
   int c   ;
   enum = nexta = nextc = numact = 0;
   numcdn = numrule = sumact = sumcdn = 0;
   noelse = nodec = noname = TRUE;
   inconsis = FALSE;
   parsact = COND;
   pbak = peek = -2;
   fptr = free;
   for (c=0; c < MAXRULE ; c++)     {
       rule[c] = rmask[c][0] = rmask[c][1] = 0;
   }
   for (c=0; c < BIGINT ; c++) {
       free[c].actptr = free[c].actltrs = 0;
}  }

run()  {             // driver for multiple table coding
   while (copycode(TOKN))  {
       reinit();
       proctab();
   }
   fflush(outp);
   exit();
}
```

```
samerule(a,b)  int a, b;   {          // test if rule a and
   char n, p;                         // rule b are identical
   int k;
   k = -1;
   while (k++ < numcdn)     {
       n = cstub[k].centry[a];
       p = cstub[k].centry[b];
       switch (n)  {
           case '-' :
               break;
           case 'y' : case '$' :
               if (p=='n'||p=='*')
                   return(FALSE);
               break;
           case 'n' : case '*' :
               if (p=='y'||p=='$')
                   return(FALSE);
   }   }
   return(TRUE);
}

setmask () {                        // set rule masks
   int i, j, orlist;
   orlist = 1;
   for(i = 0;i < numcdn;i++)   {
       for(j = 0;j < numrule;j++)  {
           orlist =<< (15-i);
           switch (cstub[i].centry[j]) {
               case 'y' :  case '$' :
                   rmask[j][0] =| orlist;
                   break;
               case 'n' :  case '*' :
                   rmask[j][1] =| orlist;
                   break;
               default :
                   rmask[j][0] =| orlist;
                   rmask[j][1] =| orlist;
           }
           orlist = 1;
}  }   }
```

```
strcopy(k,p,s) int k; char *p, *s; {   // copy string
   int i, c    ;                      // from input to p
   i = 0   ;
   while (++i < k) {
       if ((c=GNC) == DELIM)   break;
       if (c==CRLF)    line++;
       else if (c== EOF)   thatsit();
       *p++=c;
   }
   *p = NULL;
   if (i>=k && (c=GNC) != DELIM)   {
       if (c== EOF)    thatsit();
       printf(" ERROR   *S01*  ");
       printf("Invalid %s stub line %d.\n",s,line);
       killex();
}  }

sumerr(n,c) char *n, *c;   {   // print number errors
   printf(" ERROR   %s  number of %s not ",n,c);
   printf("as specified in option section.\n");
   printf("\nExecution terminated.\n");
   killex();
}

thatsit() {             //  exit for invalid end-of-file
   printf("Unexpected END OF FILE encountered, ");
   printf("execution terminated.\n");
   fflush(outp);
   exit();
}

tossline (c) int c ; {    //  toss away rest of line
   if (c!=CRLF) while ((c=gobble())!=CRLF) ;
}

totrule(n) int n;  {           // sum the number of simple
   int k, t;                   // rules in a table rule
   t=0;
   for (k=0; k < numcon ; k++) {
       if (cstub[k].centry[n] == '-')
           t++;
   }
   return(1 << t);
}    .
```

```
yyaccpt () {                    // accepted table routines
   int i;
   ptable ();
   if (enum)
       return;
   if (ambigck())
       return;
   setmask ();
   gencode();
}

yyerror(s) char *s ;   {        // syntax error handler
   printf(" ERROR   *X0");
   if (parsact == TLIST)
       printf("2");
   else if (parsact == ALIST)
       printf("3");
   else
       printf("1");
   printf("*  statement syntax line %d.\n",line);
   killex();
}

yylex() {                       // scanner
   extern int yylval;
   int c;
   switch (parsact) {           // flag for scanner
       case TLIST :
       case ALIST :
       case NUM   :
       case DIGIT :
           if((c=eatc()) >= '0' && c <= '9') {
               yylval = c - '0' ;
               while ((c=GNC) >= '0' && c <= '9') {
                   yylval = yylval * 10 + c - '0' ;
                   if(yylval > numrule) badlogic("L04");
               }
               if ((pbak = c) == CRLF) line++;
               else
                   if (c== EOF) thatsit();
               parsact = (parsact == ALIST ? NUM : DIGIT);
               return(parsact);
           }
           return(c);
       default :
           return(parsact);
}  }
```

BIBLIOGRAPHY

1. Aho, A.V., and Johnson, S.C., "L R Parsing", Computing Surveys, v. 6, p. 99-124, June 1974.

2. Davies, G. and Welland, R., "A Pre-processor Using Rule Mask Techniques for Extended Entry Decision Tables", Software, v. 3, p. 227-237, July 1973.

3. Cavouras, J.C., "On the Conversion of Programs to Decision Tables: Method and Objectives", Communications of the ACM, v. 17, p. 456-462, August 1974.

4. Ganapathy, S. and Rajaraman, V., "Information Theory Applied to the Conversion of Decision Tables to Computer Programs", Communications of the ACM, v. 16, p. 532-539, September 1973.

5. Gildersleeve, T.R., "Decision Tables and Their Practical Application in Data Processing", Prentice-Hall, 1970.

6. Johnson, S.C., "YACC - Yet Another Compiler-compiler", Bell Laboratories, not dated.

7. Kernighan, B.W., "UNIX for Beginners", Bell Laboratories, updated March 1976.

8. King, P.J.H., "The Interpretation of Limited Entry Decision Table Format and Relationships Among Conditions", The Computer Journal, v. 12, p. 320-326, November 1969.

9. King, P.J.H., and Johnson, R.G., "Comments on the Algorithms of Verhelst for the Conversion of Limited-Entry Decision Tables to Flowcharts", Communications of the ACM, v. 17, p. 43-45, January 1974.

10. King, P.J.H., and Johnson, R.G., "The Conversion of Decision Tables to Sequential Testing Procedures", The Computer Jounrnal, v. 18, p. 298-306, November 1975.

11. London, K.R., Decision Tables, Auerbach, 1972.

12. Low, D.W., "Programming by Questionaire: An Effective Way to use Decision Tables", Communications of the ACM, v. 16, p. 282-286, May 1973.

13. McDaniel, H., An Introduction to Decision Logic Tables, Wiley, 1968.

14. McDaniel, H., Applications of Decision Tables - A Reader, Brandon/Systems Press, 1970.

15. McDaniel, H., Decision Table Software - A Handbook, Brandon/Systems Press, 1970.

16. Montalbano, M., Decision Tables, Science Research Associates, 1974.

17. Muthukrishnan, C.R. and Rajaraman, V., "On the Conversion of Decision Tables to Computer Programs", v. 13, p. 346-351, June 1970.

18. Pollack, S.L., and others, Decision Tables: Theory and Practice, Wiley-Interscience, 1971.

19. Pooch, V.W., "Translation of Decision Tables", Computing Surveys, v. 6, p. 125-151, June 1974.

20. Press, L.I., "Conversion of Decision Tables to Computer Programs", Communications of the ACM, v. 8, p. 385-390, June 1965.

21. Pros and Cons of Decision Tables, Computer World, v. 7, p. 16, June 6, 1973.

22. Reinwald,.T. and Soland, R.M., "Conversion of Limited-Entry Decision Tables to Optimal Computer Programs I: Minimum Average Processing Time", Journal of the ACM, v. 13, p. 339-358, July 1966.

23. Ritchie, D.M., C Reference Manual, Bell Laboratories, 1973.

24. Schumacher, H. and Sevcik, K.C., "The Synthetic Approcach to Decision Table Conversion", Communications of the ACM, v. 19, p. 343-351, June 1976.

25. Smillie, K.W. and Shave, J.R., "Converting Decision Tables to Computer Programs", Computer Journal, v. 18, p. 108-111, May 1975.

26. Thompson, K.L., and Ritchie, D.M., UNIX Programer's Manual, Bell Laboratories, 1973.

27. Veinott, C.G., "Programming Decision Tables in Fortran, Cobol, or Algol", Communications of the ACM, v. 9, p. 31-35, January 1966.

28. Verhelst, M., "The Conversion of Limited-Entry Decision Tables to Optimal and Near-optimal flowcharts: Two New Algorithms", Communications of the ACM, v. 15, p. 974-980, November 1972.

29. Walsh, D., A Guide for Software Documentation, p. 49-60, Advanced Computer Techniques Corporation, 1969.

30. Willoughby, T.C. and Arnold, A.C., "Communicating with Decision Tables, Flowcharts and Prose", Data Base, v. 4, p. 13-16, Fall 1972.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexanderia, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Computer Laboratory, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
5.	Cdr. C. P. Gibfried, Code 52Gf Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6.	Asst. Prof. G. Barksdale, Code 52Ba Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
7.	Capt. Robert W. Roesch, Jr. 509 Willow Street Pacific Grove, California 93950	1
8.	Lt. Joseph F. Keller 1309 Croydon Street Irving, Texas 75062	1

Thesis 169900
K258 Keller
c.1 A decision logic table preprocessor.